Heterogeneous Computing with Focus on Mechanical Engineering

Jon Mikkelsen Hjelmervik

Thesis submitted in partial fulfillment
of the requirements for the degree of

Ph.D

Faculty of Mathematics and Natural Sciences,
University of Oslo

in cotutelle with

Institut polytechnique de Grenoble
l'Ecole Doctorale Ingénierie - Matériaux, Mécanique,
Energétique, Environnement, Procédé, Production
Spécialité : Génie Industriel : Conception et production

DIRECTORS OF THESIS
Jean-Claude Léon, Michael Floater, Tor Dokken

December 19, 2008

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
Nr. 842

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AiT e-dit AS, Oslo, 2009.

Produced in co-operation with Unipub AS.
The thesis is produced by Unipub AS merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Unipub AS is owned by
The University Foundation for Student Life (SiO)

Acknowledgments

Many people have contributed to this thesis. I would especially like to thank my supervisor Jean-Claude Léon for his warm welcome to France, and for the interesting discussions we have had. My supervisors in Oslo, Michael Floater and Tor Dokken have also been important discussion partners and provided valuable feedback. The discussions with my colleagues Knut-Andreas Lie and Trond Hagen throughout the entire research have been essential for my research.

There are a number of persons who I owe great thanks; Johan Seland for working with me on developing a C++ library `Shallows` together with Trond Hagen; my officemate André Brodkorb for sticking out with all my disturbances and proofreading for my thesis. Vegard Kleppe for sharing his burning passion for computer graphics and computer architectures with me. Without him I would not have been involved in this project; my fellow PhD students Robert Iacob and Rosalinda Ferrandes for taking good care of me when I stayed in France. Their help to navigate through the paperwork in a foreign country is greatly appreciated; my co-corkers at SINTEF, Vibeke Skytt, Jens-Olav Nygaard, Christopher Dyken, Sverre Briseid, Jan Thommasen, Jostein Natvik and Atgeirr Rasmussen for creating an inspiring atmosphere.

Last, but not least I would like to thank my family for their support. I greatly appreciate the support I always receive from my parents Hilde and Harald Mikkelsen. My fantastic wife Jannicke Hjelmervik supported me and listened to me talking about my research. Thank you for being you! The beautiful smiles of our daughter Alice Aurora always make my day.

Jon Mikkelsen Hjelmervik December 19, 2008

Abstract

During the past few years there has been a revolution in the design of desktop computers. Most processors today include more than one processor core, allowing parallel execution of programs. Furthermore, most commodity computers include a graphical processor that outperforms the central processor by at least one order of magnitude. Tapping into this vast resource is commonly referred to as heterogeneous computing. The change in hardware invalidates old software-design truths. There is therefore need for new algorithms, and research into adapting existing algorithms to these architectures. Our main focus has been to accelerate algorithms relevant for mechanical engineering.

In this dissertation we present four algorithms devoted to take advantage of the computational strengths of heterogeneous architectures. Each work is based on state-of-the-art hardware available at the time the research was performed.

First we describe an algorithm for high-quality visualization of parametric surfaces. This is useful in a CAD setting, were an accurate rendering is important for visual validation of model quality. We further describe simulation of shallow-water waves using a state-of-the-art numerical scheme. Our accelerated implementation gave a speedup of up to 40 times compared to an optimized reference implementation. Our implementation features real time simulation and visualization of semi-realistic nonlinear wave effects.

Finally we present two algorithms for shape simplification of 3D-models. The algorithms aim at reducing time spent on preparing models for finite element analysis. Finite element analysis is important to determine mechanical properties of objects prior to manufacture. Such analysis can be used to investigate thermal behavior and determine the strengths and weaknesses of physical components. Before the analysis can take place the models must undergo a preparation phase where shape simplification plays an important role. The first work we describe for shape simplification is a hybrid algorithm, using graphics hardware for the computationally demanding operations, and the main processor for maintaining the data structure. Our second work describes a shape simplification algorithm highly suitable for heterogeneous architectures and a reference implementation on the Cell BE.

Résumé

Au cours des dernières années, une révolution de la conception des ordinateurs de bureau a eu lieu. Aujourd'hui, la plupart des processeurs comporte plus d'un cœur, permettant une exécution parallèle des programmes. De plus, la majorité des ordinateurs comporte un processeur graphique qui dépasse en performances leur processeur central par un facteur supérieur à dix. Bénéficier de cette ressource importante est communément désigné par 'calcul hétérogène'. Cette transformation des composants remet en question les règles classiques de développement de logiciels. Notre principal objectif a donc consisté à accélérer des algorithmes présentant un intérêt dans le domaine du génie mécanique.

Dans ce manuscrit, nous présentons quatre algorithmes destinés à tirer partie des avantages des architectures hétérogènes. Chacun d'entre eux est basé sur l'utilisation de matériel le plus récent à la période à laquelle la recherche a été conduite.

En premier, nous décrivons un algorithme pour une visualisation de qualité de surfaces paramétriques. Ceci est très utile dans un environnement CAO où une représentation précise est importante pour permettre une validation visuelle de la qualité d'un modèle d'objet. Ensuite, nous décrivons une simulation d'écoulement à surface libre de vagues en utilisant un schéma numérique récent et performant. L'accélération apportée par notre implémentation produit un gain de performances correspondant d'un facteur 40 comparativement à une version de référence optimisée. Notre implémentation comporte une simulation temps réel et une visualisation d'effets non-linéaires semi-réalistes de vagues.

Finalement, nous présentons deux algorithmes appliqués à la simplifications de formes 3D d'objets. Les algorithmes sont destinés à réduire le temps de préparation de modèles éléments finis. Une analyse par éléments finis est une étape importante pour caractériser les propriétés mécaniques de composants préalablement à leur fabrication. Une telle analyse peut être utilisée pour étudier un comportement thermique et évaluer les forces et les faiblesses de composants réels. Avant que cette analyse puisse être conduite, les modèles doivent subir une phase de préparation où la phase de simplification de forme joue un rôle important. Le premier de ces deux travaux que nous décrivons est un algorithme hybride de simplification de formes, utilisant une carte graphique pour les calculs les plus intensifs et le processeur central pour assurer la cohérence de la structure de données. Le second travail décrit un algorithme de simplification de formes particulièrement adapté à des architectures hétérogènes et son implémentation sur une architecture cellulaire de type 'Cell BE'.

Contents

Introduction

During the last few years the trend in commodity hardware have gone from single-core processors to homogeneous multi-core or heterogeneous many-core processors. This evolution is driven by difficulties in developing faster cores as well as the evolution of highly parallel processors, e.g., Graphics Processing Units (GPUs).

The development in hardware has made research in new algorithms necessary, since not all algorithms are well suited for the new massively parallel architectures. During the past years the interest for such research has increased tremendously, and has grown into a new discipline in computer science called "General-Purpose Computing on GPUs" (GPGPU). Research in GPGPU has attracted the interest from researchers all over the world. Due to the increase in flexibility of GPUs more and more algorithms can be adapted to take advantage of them. The research has led to development of new algorithms as well as new uses of algorithms not commonly used. Bitonic sort by Baxter [6], which has been successfully implemented on GPUs by Purcell et al. [54], is one of many examples. For a thorough overview of the field we refer to Owens et al. [52].

The goal of our research has been to investigate the usefulness of GPUs in numerical applications. The research presented here started in 2004. At that time there was virtually no research available due to the lack of programmable GPUs with floating-point capabilities. As part of our research we developed a programming library called `Shallows` [34] aiming at making GPGPU programming easier and safer. The GPU-based implementations presented in this document use `Shallows` or its predecessor.

Recently, graphics vendors have released programming tools that provide access to the GPU directly, bypassing the graphical system. Such tools are also available for GPU-based accelerators without ability to output images, dedicated to numerical computations. The evolution of software as well as hardware is narrowing the gap between GPUs and CPUs making GPGPU more feasible.

Another trend in processor design is heterogeneous processors such as the *Cell BE* processor from Sony, IBM and Toshiba. The Cell BE consists of one traditional CPU core and eight *thin* cores with reduced functionality. The thin cores are simple cores dedicated to computations. In contrast to GPUs, where all cores operate in operate synchronously, the cores in the Cell BE operate independently, and can even execute different programs. This places the Cell BE between current GPUs and multi-core CPUs when it comes both to flexibility and performance. Our most recent work regarding shape simplification requires functionality that was not available on the GPUs available at the time we designed the algorithm and performed the Cell BE implementation. Hopefully, we will develop an efficient GPU-based implementation at a later time, allowing us to validate if the algorithm also is suitable for GPUs.

In our work we have focused on three different applications, namely visualization of parametric surfaces, physical simulation and shape simplification. The applications were chosen both because of the need for efficient software and due to their suitability for GPU-based implementations.

The first application we targeted was high-quality rendering parametric surfaces. Visual inspection of surfaces is important when validating the quality of CAD surfaces. In this process the goal is not to produce a visually pleasing image, but rather to produce a correctly rendered surface. Focusing too much on performance has a tendency to lead to loss of important details. However, the application should be able to render surfaces both from a distance and close-up at interactive frame rates. We developed a tessellation/rendering algorithm called "GPU-based screen space tessellation" that use the GPU both for tessellating and rendering parametric surfaces. We implemented this algorithm for cubic tensor product B-splines.

Next, we studied acceleration of physical simulation using GPUs. Physical phenomena are usually modeled by PDEs that are difficult, or impossible to solve analytically. Therefore, numerical methods are used to compute approximate solutions for the simulation. Faster computers allow larger problems and increased accuracy of the solution without increasing the computation time. It is therefore important to take advantage of the computational resources available. Therefore, we studied acceleration of physical simulation using GPUs. We focused on hyperbolic conservation laws, because most high-resolution schemes for systems of hyperbolic conservation laws are explicit and numerically stable. Therefore, simulation based on hyperbolic conservation laws are well suited to be implemented on GPUs.

The third application we studied is mesh simplification. Simplification of triangulations is often used in computer graphics and plays an important role in preparing CAD models for Finite Element Analysis (FEA). A Finite Element mesh (FE mesh) is usually built from a triangulation, originating either from a range scan or a CAD system. In both cases the triangulation includes shape details that should be removed before the generation of the FE mesh. This is a time-consuming task that requires interaction with the user to verify that the simplified model is satisfactory. Improving the performance of mesh simplification is therefore of great importance to reduce the total time spent preparing a 3D model for FEA.

This document consists of two parts, background information and our contribution. Chapter 1 gives an introduction to parallel computer architectures currently available as commodity hardware. The architectures described are multi-core processors, the Cell BE processor and GPUs. Programming tools for GPUs are discussed in Chapter 2. Chapters 3 trough 5 present background information to the applications we targeted and presents our problem statements. Our contribution is presented in Part 2, where Chapters 6 trough 9 present the four algorithms we developed. Chapter 10 summarizes the contributions of our work and gives our outlook into the future of heterogeneous computing.

Part I

Background

Chapter 1

Parallel Architectures

Supercomputers have been based on parallel architectures with distributed memory for decades. This allows programs to solve problems that do not fit within system memory of a single node, without using virtual memory. Furthermore, it allows the computational power of large clusters to solve large scale problems in reasonable time. Clusters are usually built from a set of computer nodes, each containing one or more CPUs and a high speed network connecting the nodes. Multi-CPU solutions have been available for desktop computers as well, but not for the commodity market. They have been too expensive and complex for most users.

Moore's Law, by Intel co-founder Gordon Moore, predicts that the number of transistors that can be placed inexpensively on an integrated circuit doubles every second year. This prediction is tightly linked to the exponential increase in clock frequency and the total performance increase we have seen until recently. Even though Moore's Law still applies, performance of single-core processors is not increasing as fast as predicted. Since microprocessor manufacturers are unable to transform the increased transistor count into processing power in single-core CPUs, they have turned to multi-core CPUs instead. Multi-core CPUs consume less power than single-core CPUs with the same theoretical performance, making multi-core an interesting technology for notebooks and handheld devices.

Both single-core and multi-core CPUs use most of their transistors on cache and logic. This is useful for algorithms with a high number of branches and unstructured memory accesses. However, this design is not so well suited for parallelizable applications with high computational density. Architectures with more primitive cores allows a higher number of cores and thus more floating point operations per second within the same transistor and power budget. Examples include GPUs and the Cell BE. These processors have a high number of thin cores, that are controlled by more traditional CPU cores. The Cell BE is a heterogeneous multiprocessor, with both thin and traditional cores. A GPU is a homogeneous processors used to accelerate applications executed on a CPU. A system consisting of CPU(s) and GPU(s) is considered a heterogeneous system since it features different processing units.

1.1 Brick Wall for Serial Computing

Even though the number of transistors that can feasibly be integrated in a CPU and the transistor density are increasing, the performance increase has lost its pace. In the period from 1978

to 2002, the yearly performance increased for single-core processors was approximately 50%. Since then, it has dropped to less than 20%. This is due to difficulties related to memory latency, Instruction Level Parallelism (ILP) and power consumption. Asanovic et al. [2] introduced *Power wall + Memory Wall + ILP wall = Brick Wall*, meaning that it seems unlikely that we will overcome these problems using serial architectures. The following sections give a short description of each wall.

1.1.1 Memory

In the early days of computing, computers were developed as one, and hence more balanced in performance. As each component has been optimized and developed further, this has changed. Memory modules used in commodity computers are not able to keep up with the CPUs. This is known as the von Neumann bottleneck, a term that was introduced by Backus [4] already in 1977. Even more important than the bandwidth between memory and CPU is the increasing latency. Today, the time between a CPU issues a memory request until the data has reached the CPU can be several hundred clock cycles. This relative latency has increased steadily, and it is not expected that the development of memory modules will narrow the gap in the near future.

Modern CPUs have large caches to remedy the relatively slow memory. In order to realize the performance increase expected by higher clock frequency, the caches has grown substantially from year to year. Now the processors have reached such high speeds that it is difficult to obtain the expected performance increase with this strategy.

1.1.2 Instruction Level Parallelism

The development of processor design has lead to increasingly complex processors in order to optimize the number of instructions that are executed per clock cycle. Much work has been done to increase the parallelism at instruction level by simultaneously performing independent operations.

A superscalar CPU executes several instructions during each clock cycle by dispatching multiple instructions. Typically, superscalar CPUs are also pipelied, such that each instruction is broken down to a number of simpler operations. This allows the different units (arithmetic, logical, etc.) to operate on different stages in the pipeline, and thus operate on one of the dispatched instructions.

Dependencies between the instructions can prevent efficient utilization superscalar CPUs. To increase the instruction throughput, some CPUs change the order instructions are dispatched in. This concept is called *out-of-order execution*. This is especially useful for deeply-pipelined superscalar processors, as it can delay the execution of an instruction that is dependent of another instruction still in the pipeline. Efficient reordering of instructions is a complex task that requires a large number of dedicated transistors.

As the pipeline length increases and the longer the CPU must wait for memory fetches, the more expensive branch prediction misses become. Correct branch prediction therefore becomes essential to take advantage of ILP. However, even though the CPU designs are improving, we cannot expect the same performance increase rate as seen before. The sequential nature of the instruction stream makes it increasingly difficult to improve the parallelism at runtime.

1.1.3 Power

The power consumption of a processor is tightly linked to its clock frequency. This not only increases the power bill, but also leads to a challenge when it comes to heat dissipation. Modern CPUs are able to reduce the power consumption when they are not fully used, thus also reducing the production of heat. However, it is not likely that the peak power consumption of commodity processors in the foreseeable future will become significantly higher than it already is.

When altering the clock frequency of a CPU, the change of power consumption is much higher then the change in performance. Therefore, underclocking a CPU improves its performance to power ratio. Consequently, a multi-core processor has higher performance compared to a single-core processor with the same power consumption. The power consumption is one of the most important motivations for multi-core processors today. AMD, one of the major CPU vendors, has stated that they do not plan to release processors with higher power consumption than today.

1.2 Shared Memory Architectures

The "brick wall for serial computing" has paved the ground for one of the largest changes in the computer industry. Virtually no new processor architecture is based on single-core designs. A common denominator for the new designs is that an increasing number of parallel units are connected to shared memory. Even though there is a large difference between the GPUs and multi-core CPUs, they share some common features related to the shared memory model. We will here describe some key properties of shared memory architectures and some topics related to thread safe algorithms. An algorithm is said to be *thread safe* if it behaves correctly when executed in parallel.

Algorithms written for distributed systems must focus on reducing the intra-node communication to a minimum. In particular, large amounts of data should not be transferred unless strictly necessary. In a shared memory system the processors can read and write to the same memory, eliminating the need for intra-processor transfers. Therefore, algorithms developed for distributed systems may not be efficient when used on a shared memory architecture.

Most processors use caches to communicate with the shared memory. A cache can be shared among all units associated to the memory, or local associated to one or a group of parallel units. In the presence of several local caches data integrity issues arise, usually referred to *cache coherency*. Architectures equipped with special hardware to resolve these issues are referred to as coherent caches. From a programmers point of view, coherent caches are indistinguishable from one single cache. An algorithm to be executed using incoherent caches, however, may need adaptation to each architecture it is targeting to take advantage of how the cache behaves.

Exclusive access to a shared resource are often required in a parallel setting. Commonly, mutual exclusion, also known as *mutex*, is used to facilitate exclusive access. A key feature of a mutex is that the process of reading its state, updating it and writing it back to shared memory is performed without interruption by competing parallel units. Such sequence of operations are known as an atomic read-update-write operation, and is a special case of *atomic operations*. Atomic operations refers to a set of operations that appears to the rest of the system as one operation. Most parallel architectures provide atomic read, write and read-modify-write, and

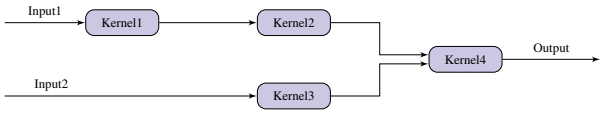


Figure 1.1: A pipeline of four kernels transforms the two input streams into one output stream.

Listing 1.1: Stream programming example

```
setInputArrays(A, B);
setOutputArrays(C);
loadKernel(''out = input1 + input2'');
execute();
```

other atomic operations that can be built from these. Since loading and storing data to and from registers normally are single instructions, these operations are normally atomic as long as one uses basic data types.

1.3 Stream Programming Model

Many applications can be split into a number of independent tasks that can run independently. This opens up the possibility to execute the tasks in parallel, this strategy is called *task parallelism*. The number of independent tasks limits the number of parallel units, making it difficult to write applications that take advantage of architectures with many parallel units. Performing one task in parallel for different data elements on the other hand is called *data parallelism*. Algorithms that are implemented in a data parallel fashion are usually independent of the actual number of cores. Provided the number of data elements is (much) higher than the number of cores, the performance of such implementations will normally scale well with increasing number of parallel units.

Many data parallel algorithms may be organized as computational kernels operating on a data stream, as illustrated in Figure 1.1. In the *stream processing model* the programmer specifies a *kernel* and a data stream. A kernel is a small program that is repeated for each successive element of its input streams, computing its output stream(s). Since the kernel operates solely on the input streams it is not possible to do in-place computations. However, this restriction allows the kernels to operate in a data-parallel fashion.

Data processing is traditionally based on an instruction-driven model. This model corresponds to the von Neumann architecture. In this architecture, a single processor executes a single instruction stream to operate on data stored in the same memory as the instructions. The data needed for execution of an instruction are loaded into the cache during the processing. This makes instruction driven processing flexible but very inefficient when it comes to uniform operations on large data blocks.

The stream programming model can be used as an abstraction to the underlying hardware, allowing the programmer to develop hardware independent code. Then, the same source code can be used both for multi-core CPUs and special processors designed for the stream programming model. As illustrated in Listing 1.1, the loop iterating over the data elements is not explicitly in the code. Instead, the kernel is implicitly called for each data element.

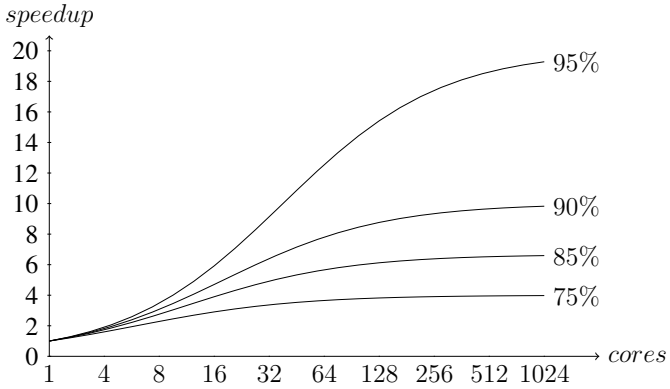


Figure 1.2: Amdahl's law. The speedup of a applications with increasing number of cores. Each graph represents an application with a given portion that can be parallelized.

1.4 Multi-Core Processors

Multi-core CPUs combine two or more traditional processor cores into a single silicon die. The main advantages of multi-core versus multi-CPU systems are shorter distance between the cores, and that the motherboard design is simpler. Today, commodity hardware allows single-, dual-, and quad-core CPUs to be used on the same motherboards, making the upgrade simple from a hardware point of view. The inter-core communication is kept within the same physical package easing the motherboard design, since it only needs to handle the communication between the CPU and the rest of the system.

From a user's point of view there is no large difference between using multiple CPUs and multi-core CPUs. Desktop computers can take advantage of dual-core systems without modifiacions to the software, as long as the operating system supports the processor and multi-tasking. This is due to the fact that desktop computers usually runs a number of processes in the background (firewalls, disk indexing etc.) and adding a second core can execute these processes.

Reducing the clock frequency and increase the number of cores leads to a higher performance for the same power consumption. The question is then how low should the clock frequency be and how many cores should one use? The answer to this is partly linked to the particular applications and how large parts of the applications we are able to parallelize. Reducing the clock frequency will decrease the performance of sequential code, and the speed of a single core will become the limiting factor. Amdahl's law [1] gives the optimal theoretical speedup for problems with known parallel fraction. The law is written

$$\frac{1}{(1-P)+\frac{P}{S}},$$

where P is the fraction of parallizable code, and S is the number of cores. This gives a upper bound on how much performance can be gained by increasing the number of computational cores when P is known. As illustrated in Figure 1.2, increasing the number of cores beyond a certain threshold gives almost no performance gain, converging asymptotically towards $\frac{1}{1-P}$.

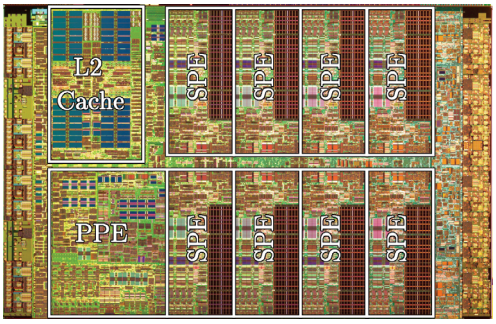


Figure 1.3: Cell BE die photo. The different cores and the PPEs cache is marked.

1.5 Heterogeneous Architectures

We have seen that multi-core CPUs make good sense when it comes to performance per watt, compared to single-core CPUs. Furthermore it is a way to take advantage of the increased number of transistors that we are able to put into a single package. However, the sequential parts of the application can easily be the bottleneck when there are no very fast cores. This advocates a heterogeneous architecture, where we have a few traditional cores (*fat cores*) for the sequential parts and a high number of reduced functionality cores (*thin cores*) for the data-parallel parts.

The thin cores do not need to run any operating system, but can be managed by the fat cores instead. Furthermore, the data parallel parts that can take advantage of a very high number of cores can often be implemented with relatively simple code. This implies that the instruction set of the thin cores can be reduced in addition to the cache sizes and logic related to out of order execution.

Heterogeneous architectures come in a wide variety of forms, the two most widespread are the Cell BE processor and GPU-CPU systems. The Cell BE contains both a fat core and eight thin cores in one single physical package. GPUs on the other hand currently consist of several hundred thin cores, and must be controlled by a CPU.

1.6 Cell BE

The IBM Cell Broadband Engine Architectures is a heterogeneous processor developed by IBM, Sony and Toshiba, targeting both supercomputers and the home entertainment market. In addition to one or more traditional processor core(s), called Power Processing Elements (PPEs), the Cell BE includes a number of thin cores called Synergistic Processing Elements (SPEs). The PPE is based on the Power Architecture from IBM and contains both L1 and L2 cache. The SPEs can be considered as co-processors to the PPE and contain a local store.

Instead of a hardware controlled cache, each SPE has 256KB of local store, used to store both code and data. A program running on a SPE does not access system memory directly, but uses Direct Memory Access (DMA) instructions to copy data to and from system memory. DMA transfers are asynchronous, allowing the SPE to continue execution while data is being

transfered. DMA instructions can also be used to transfer data between the SPEs local stores.

1.6.1 SPE Computational Performance

The computational strength of the Cell BE comes from the high number of SPEs. Hardware for out-of-order execution and branch prediction has been sacrificed to keep power consumption and transistor count down. This design allows exact performance prediction, which is important in many realtime multimedia applications. Parallel sections of video decoding is an example that is well suited for implementation on the Cell BE.

Similar to modern CPUs, the SPEs have a vector unit for 128bit long vectors, and are able to issue one fixed/floating point operation on such vectors per clock cycle. This gives the SPEs in a current Cell BE running at 3.2GHz a total peak performance of 102 GFLOPS for double precision and 204 GFLOPS for single precision. This performance is comparable to the fastest octa-core CPUs commercially available at a much smaller power consumption and transistor count.

Where modern CPU cores use sophisticated logic to make unoptimized programs run fast, this is left to the compiler (and programmer) when developing software for the SPEs. The benefit from techniques like loop unrolling is therefore higher on the SPEs compared to todays commodity CPUs.

1.6.2 Data Transfer

From a programming point of view, memory management related issues constitute the main differences between traditional homogeneous architectures and heterogeneous processors such as the Cell BE.

In addition to DMA instructions, each SPE has an atomic unit, providing atomic read-update-write operations. The main purpose of this unit is to allow atomic operations such as mutexes. The atomic unit can also be used to create a software-based coherent cache, as the Cell BE is not equipped with a hardware cache. However, one should try to reduce the use of the atomic unit, as it easily can become a bottleneck in the application. A software managed cache can also use DMA transfers to and from local store. Such a cache normally contains functions to read/write data, initiate reading of a memory location (touching), and notifying to the cache that the content may be outdated (dirtying).

An implication of lacking coherent cache is that we have no guarantee that memory transfers are performed in the order they are issued. The Cell BE however has an option to obey the order of the issued commands. Each function to initiate data transfers using DMA has the option to postpone the DMA transfer until other DMA transfers have completed. By using this option the transfer is delayed until other transfers initiated from the same SPE have completed. The Cell BE is optimized to handle a large number of DMA transfers in parallel, therefore it can be beneficial to avoid this kind of serializing of the data transfers.

The cost of a cache-miss is reduced if the processor is busy while the memory transfer takes place. This can be achieved by touching the cache before the data is needed. This strategy is only feasible if the program can continue execution before the data is transferred. Bader et al. [5] presented an alternative latency-hiding technique for the Cell BE. They use software-

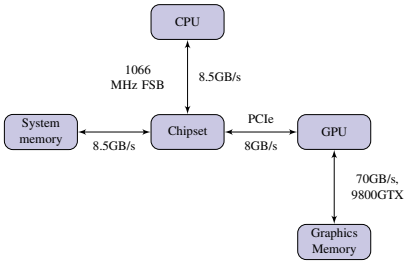


Figure 1.4: The overall system architecture of a typical PC.

managed threads to let the SPE continue working after a memory request is issued. The SPEs do not have hardware support for quickly switching between threads. Therefore, the program itself is responsible for switching between the software-managed threads.

These strategies may be combined by *touching* a cache location before switching software thread. Due to the performance overhead involved in switching threads and using a software cache, both strategies may also decrease the performance for some applications.

1.6.3 Roadmap

The Cell processor has been commercially available in large volumes in the PlayStation 3 video game console since November 2006. This version was produced using a 90 nm process and did not feature hardware for high-performance double precision. In May 2007 IBM announced a 65 mm version with a theoretical performance of 102 GFLOPS in double precision. This version is used in the IBM supercomputer Roadrunner, which *top500.org* [63] reported to be the world's first petaflop/s computer.

As the production process continues to shrink, IBM plans to release more power efficient versions of the Cell BE. There are also official plans to make a version with 34 cores. This version will feature 32 SPEs and 2 PPEs, aiming at delivering 1 TFLOPS.

1.7 Graphics Processing Units

Most modern PCs have programmable graphics processing units (GPUs). Such GPUs typically give a floating-point computational power that is more than one order of magnitude higher compared to the CPU in a modern PC. While CPUs are instruction driven, GPUs are data-stream driven. This means that the GPU executes the same instruction sequence on large data sets. The instruction sequence to be executed is uploaded to the GPU, before the execution is triggered by a data-stream. The result of the computation can then be used for visualization, processed by a new instruction sequence, or read back to the CPU.

In contrast to the Cell processor, a GPU is treated as a single co-processor and the parallelization is performed implicitly within the GPU. This fact makes the *stream processing* programming model described in Section 1.3 well suited for GPGPU applications.

GPUs are often placed on a separate graphics cards together with dedicated graphics memory. As illustrated in Figure 1.4, the Front Side Bus (FSB) connects the CPU to the chipset.

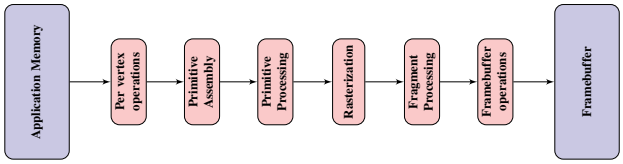
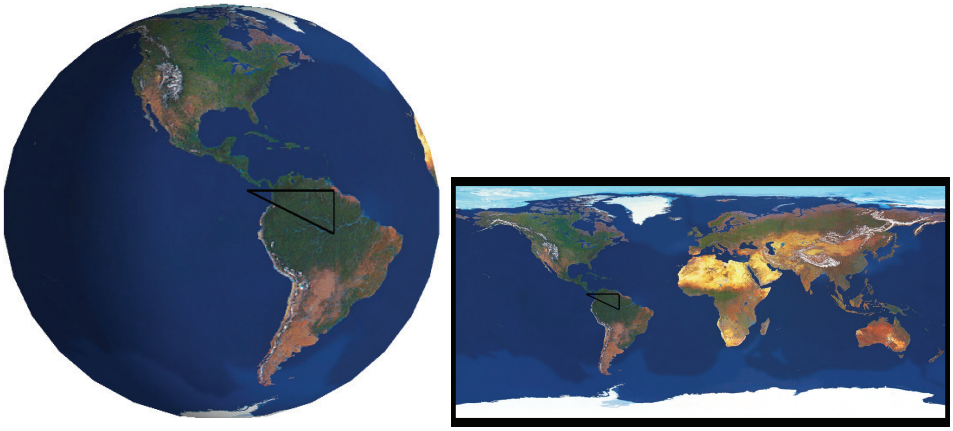


Figure 1.5: Simplified view of the graphical pipeline.



(a) Textured sphere. (b) Texture illustrating the earth.

Figure 1.6: Figure (a) shows a textured sphere where one triangle is marked with a black line. Figure (b) shows the texture used, and the corresponding triangle.

The chipset links the CPU to the system memory, GPU and other peripherals. The PCIe bus is used to transfer data between chipset and graphics memory. This data bus can easily become the limiting factor if the work to be performed by the GPU is too small to hide the cost of the data transfer.

GPUs come in many flavors and are currently undergoing large architectural changes. We will therefore give a short description of three different generations of GPUs, starting with the fixed function pipeline (used before the term GPU was introduced).

1.7.1 Fixed Functionality Pipeline

Originally, graphics accelerators used dedicated hardware for each stage in the graphics pipeline (illustrated in Figure 1.5). As the stages were implemented individually they operated in a task parallel pipelined fashion. Each stage is implemented in a data parallel fashion, yielding very high performance.

The first stage in the pipeline is the *per-vertex operations* also known as the *transform and lighting* stage. The main operations at this point are transformation of vertex positions and lighting calculations to compute the vertex's color. Then the vertices are transformed, collected to primitives which in turn are rasterized. Rasterization is the process of decomposing graphics primitives into smaller elements corresponding to pixels in the output buffer. The

elements generated by the rasterization process are referred to as *fragments*. Vertex attributes, e.g. color, are linearly interpolated over each primitive. During the rasterization the midpoint of a fragment is used for testing if the fragment is within a primitive. The midpoint is also used when evaluating the interpolated vertex attributes. A number of operations take place on fragments after the rasterization. This collection of operations is called the *fragment processing* stage. In this stage each fragment is treated independently, and the fragments color is computed. There is one type of operations in this stage that is more important than the others, namely texturing operations.

Textures are used to specify the color of a fragment. This is done by creating one, two, or three-dimensional images containing the color details, and using the interpolated texture coordinates (from the rasterization stage) for texture lookup. Figure 1.6 shows such a mapping from a 2D image to a 3D sphere. The process of attaching an image to geometry this way is often referred to as *texture mapping*. Texture mapping enables a vast variety of effects, and many extensions to the fixed functionality pipeline have been defined to make the texturing more flexible. It is in this part of the pipeline most where GPGPU calculations are performed.

The last stage in the pipeline before the final pixels are stored in the frame buffer is called *frame-buffer operations*. Some of these operations are relatively simple and are normally implemented very efficiently in hardware, hence utilizing them for GPGPU purposes can be highly efficient. This includes operations like *alpha test*, *stencil test*, *depth test*, and *blending*.

The alpha test is used for accepting or rejecting a fragment based on its alpha (opacity) value. The stencil test compares a reference value with the value stored in the stencil buffer. Depending on the result of the test, the value in the stencil buffer is modified. The depth test is used to decide if the fragment should be drawn by comparing the depth (distance from the viewpoint) of the incoming fragment to the depth stored in the frame buffer. Blending is used to blend the color of the incoming fragment with the color stored in the frame buffer, based on current blending states.

The fixed functionality and specialized hardware resulted in very high performance at the expense of flexibility. The restricted functionality and only support for 8bit fixed point operations of commodity accelerators made them unfeasible for most general purpose computations. However there were some early GPGPU works using hardware generation, for example Harris et al. [30], and Larsen and McAllister [43].

1.7.2 Programmable Pipeline

As the performance of graphics accelerators increased, more and more features were implemented in hardware, allowing game developers to increase the complexity of their 3D graphics. Together with the flexibility, also the complexity of both hardware and APIs increased. This lead to fully programmable stages, first in the vertex stage was replaced by a programmable vertex processor. Later the fragment stage was replaced by a programmable fragment processor. Generally, the load is much higher for the fragment stage, therefore more hardware resources were spent on this stage of the pipeline. All other parts of the pipeline remained unchanged.

A program intended to run on graphics hardware is called a shader. There are two types of shaders; *vertex shaders* and *fragment shaders* intended to run on the vertex processor and the fragment processor, respectively. Both the vertex and fragment processor operate in a data-

parallel fashion, similarly to how vector processors operate. The units working in parallel are tightly coupled, and are not to be regarded as regular computational cores.

The parallel units in GPUs currently available operate in a synchronized manner. All synchronized units execute the same path, and thus if an if/else structure is used, there are cases where the time required to execute the shader is the same as the time required for executing both branches. This overhead is avoided if fragments located near each other follow the same execution path, since the different parallel units are evaluating neighboring fragments. Originally, all the parallel units of a GPU where synchronized. Currently it is common that only groups of parallel units operate synchronized.

The programmable fragment processor opened up the possibility to use GPUs for a wide range of applications. In Dokken et al. [14] we gave a detailed description of the use of GPUs for numerical computations and presented our works in this field. Owens et al. [52] gives a comprehensive overview of the state-of-the-art (2008) in the field of GPGPU.

1.7.3 Unified Shader Model

Before the unified shader model was introduced it was not possible to skip any of the stages in the graphics pipeline. Therefore, all algorithms had to be reformulated into graphical primitives and "rendered". This imposed some overhead and made GPU programming cumbersome.

As more programmable stages were introduced in the pipeline and the flexibility of each stage was increased, it became natural to use the same arithmetic units across the pipeline. This led to a unified shader model, where all programmable stages used the same units and (mostly) the same instruction set. Dynamic load balancing is used to allocate the "correct" number of arithmetic units for each stage in the pipeline. Thus, most applications benefit from this architecture, whether they are fragment shader intensive applications like most GPGPU applications, of vertex shader intensive. The unified shader model was introduced together with the possibility to store the output stream from a vertex shader in a buffer in graphics memory. This is known as *stream out* allowing the resulting vertices to be used in multiple rendering passes. In this case, there are no fragment shader active, and vertex shader can utilize all the computational power of the GPU.

In some cases, groups of GPU work items are able access a small shared memory. This shared memory can be used to create a software managed cache, and thus reduce the amount of data transferred between the graphics memory and the GPU. Furthermore, this opens up the possibility of fast communication between otherwise isolated parallel units. This is in contrast to the local store in a Cell BE's local store, which is not shared among the SPEs.

When GPUs based on the unified shared model became available, the first true GPGPU accelerators were introduced. These were graphics cards with extra memory but without video output. The accelerators are available both as rack-mounted enterprise servers and as accelerator cards for PCs. These systems are programmed using GPGPU APIs such as NVIDIA's CUDA and AMD's CAL.

Chapter 2

APIs for GPGPU

When the programmable GPUs became available, the GPU vendors did not provide high level APIs suitable for general purpose computations using GPUs. The only way to take advantage of the programmability of GPUs was by using graphics APIs. This made it rather cumbersome to use the GPUs for general computations, as applications directly using graphical APIs were difficult to read and maintain with even simple data flows. A higher abstraction layer was therefore needed. Applications rendering complex scenes usually use a *scene graph* to handle rendering-related tasks. They represent an intuitive abstraction to the graphical system, and can also be extended for GPGPU purposes. However, scene graphs focus on rendering 3D objects, not computing on data streams. Therefore, they are less suitable for GPGPU, since these applications do not require complex scenes to be rendered. Instead, the shaders and the data flows in between are often complex in GPGPU algorithms. Consequently, there was a need for new APIs specially designed for GPGPU. Such APIs could ease the burden of developing software that uses the computational resources of the GPU.

If the result of a computation is to be visualized, it is beneficial to keep the data in graphics memory. Thus, for applications using the GPU for both computation and visualization it is important that the GPGPU-API either can perform rendering or share data with graphical APIs.

2.1 BrookGPU

Brook is a programming language developed for the stream programming model, and BrookGPU is a compiler for GPU programs. BrookGPU was one of the earliest high level development tools for GPGPU applications. Developers who are more familiar with stream processing than computer graphics are the main users of this language. It extends ANSI C to allow data parallel operations, uses a preprocessor to generate C++ code, where the computational kernels are translated into fragment shaders. Brook is also capable of generating code to be used on special stream processors. More recently, BrookGPU has been extended to Brook+, an implementation by AMD, currently only available for AMD GPUs.

2.2 Sh

Another early work is Sh, primarily aiming at easing development of shaders. Sh can also be used to generate fragment-and vertex shaders used during rendering. Sh evolved into a C++ library for the stream programming model. Similar to BrookGPU, the user does not need any knowledge of the underlying graphical API to use the library. Today, Sh is no longer maintained but is replaced by RapidMind, maintained by spinoff company of the Sh project. RapidMind is now an API for the stream programming model, and can use the same source code for GPU, Cell BE and multi-core CPUs.

2.3 Shallows

To meet the need of a new GPGPU-API, we developed the `Shallows` [34] library. `Shallows` is a joint work between Jon Hjelmervik, Johan Seland, and Trond Hagen. A large portion of the development time for our first GPU applications were spent on developing `Shallows`. The library was designed to make GPGPU programming easier and safer, at a time when the available tools were highly limited. The goal was to reduce the time spent on developing OpenGL related source code, allowing developers of GPGPU applications to concentrate on implementing the algorithms instead. GPGPU applications are all about performance, the overhead by using `Shallows` is therefore kept low. A short description of the programming library, together with a usage example is given in Appendix A.

2.4 CUDA

After NVIDIA released GPUs based on the unified shader model, they also presented a dedicated GPGPU programming language called CUDA. This programming language allows the user to take advantage of hardware features not available during normal rendering. These features include access to shared memory and atomic operations. Together with a compiler, the CUDA SDK comes with an API to initiate computations on the GPU and transfer data to and from the graphics memory. CUDA is designed for computations only and does not have the capability to render to a screen. However, it is possible to share data between CUDA and graphics APIs.

2.5 OpenCL

OpenCL is a programming language not unlike CUDA, developed by a group of software and hardware companies, initiated by Apple. It is expected to be accepted as an open industry standard for GPGPU software development, supported by all major graphics vendors. This is in contrast to CUDA and Brook+ which are only supported by NVIDIA and ATI respectively. At the current time the available information concerning OpenCL is limited, because it is not yet formally accepted. It is expected to be available at the end of 2008.

Chapter 3

Visualization of CAD-Type Surfaces

CAD models are usually represented using NURBS surfaces, and subdivision surfaces are popular in the entertainment industry. Yet the GPU does not support rendering of either surface type. Instead, the surfaces must be tessellated (converted into sets of triangles) by the application before being sent to the graphics subsystem. Applications using dynamic objects and those allowing extreme close-ups of static objects require reevaluation of the surfaces at runtime.

3.1 Problem Statement

Visual inspection of surfaces is important when validating the quality of CAD surfaces. The standard method for visualizing a parametric surface is to tessellate it, uniformly or non-uniformly, and render the triangulation. The rendering system performs lighting calculations using vertex positions and normals given as input. The color values are then linearly interpolated across each triangle. A more advanced method is to use instead interpolated surface normals, and use programmable fragment processors in order to perform the lighting calculation per pixel. This strategy produces higher visual quality, but the correct color is still only obtained at the vertices.

To improve the quality, the surface normal can be evaluated per pixel. Per pixel normal evaluation requires that the parameter value of each vertex is given as a texture coordinate. This is possible for various types of surfaces, including B-spline and subdivision surfaces [8]. While this improves both the visual quality and the correctness, we still have the problem that only the vertices are rendered correctly. To further improve the correctness it is necessary to use a tessellation where each rendered triangle only covers a few pixels on the screen. There is always a tradeoff between performance and correctness in the visualization. The goal for our research was to develop a rendering method with realtime performance on a NVIDIA Geforce 5 series GPU, where the triangle size remains constant independent of viewing distance and orientation.

3.2 View-Dependent Tessellation

At the time we developed our screen space tessellation method, there existed numerous methods for view-dependent tessellation of parametric surfaces. These include hierarchical methods such as quadtree-based triangulations [48] and progressive meshes [35]. Common to all methods is that the tessellation is computed on the CPU. A copy of the tessellation is kept in graphics

memory, and must continuously be updated. Such methods may cause high CPU use, and the bandwidth for data transfer to the GPU may further limit the frame rate. As illustrated in Figure 1.4, the bandwidth between the GPU and graphics memory is of an order of magnitude higher than any other data bus in a typical PC. Keeping the triangles in GPU memory is therefore preferable.

Evaluation of subdivision surfaces with pre-evaluated basis functions utilizing a GPU is described by Bolz and Schröder [8]. The basis functions and the control points are stored in textures, and fetched by the fragment shader, where the surfaces are evaluated. The method is generalized to all surface types which can be written as linear combinations of basis functions. Our implementation of surface evaluation is based on their work.

Geometry images were introduced by Gu et al. [49] as a method for representing triangulations as images. To create a geometry image, the triangulation is cut such that it becomes topologically equivalent to a disk. This cut version is then parameterized onto a rectangle before being sampled, one sample per pixel in the image. This representation allows the triangulation to be stored without any topological information.

A similar technique is that of Multi-chart geometry images by Sander et al. [58]. The triangulation is cut into several charts, such that each chart is topologically equivalent to a disk. Each chart is then parameterized and sampled before the charts are rasterized and stored in a texture. When reconstructing such an image G^0 continuity holds only within each chart. Therefore zippering steps are necessary to ensure that the reconstructed triangulation is "watertight".

Guthe et al. [25] presented a method for rendering trimmed T-spline surfaces. In their work, the CPU determined the appropriate tessellation level and the vertex shader evaluated the surface. Predefined grids with appropriate topologies were defined for each level of detail, such that the vertex shader could evaluate the surface. Because of hardware restrictions at the time, the surface was split into a set of bicubic Bézier patches which can be evaluated at the vertex processor. The unified shader model lifted this requirement. The irregular data nature of T-splines, however, still makes it interesting to split the surface into more regular structures before rendering.

3.3 Per Pixel Correct Rendering

Yasui and Kanai [38] proposed a method for using the surface evaluation capabilities of modern GPUs for correct rendering of surfaces. Loosely speaking their method renders a coarse tessellation of the surface using a fragment shader that evaluates the surface using the linearly interpolated parameter values, and "moves" the fragment into the correct position of the frame buffer. Since the fragment processor did not have the freedom to choose which pixel it is writing to, i.e. moving the fragment, the algorithm is split into two steps.

First, the tessellation is rendered into an off-screen buffer, storing the parameter values linearly interpolated across each triangle. Second, the positions and surface color are calculated and stored in separate off-screen buffers, which in turn are converted into vertex arrays. The vertex arrays are then rendered as points. Since one of the vertex arrays contains the reevaluated positions, the points are drawn into the correct positions in the frame buffer.

The result is an image where each pixel is colored based on the "moved" position and the

surface normal evaluated accordingly. As the final image is generated by rendering points, it may contain holes. The solution proposed is to enlarge the off screen buffers used, which in turn will increase the number of points drawn. This will reduce the probability of holes, but the amount of enlargement needed is not clear. Only points originating from parameter values of visible geometry is rendered. This may in some cases cause incorrect rendering or holes in the geometry. Our contribution is inspired by this work, and we developed a visualization algorithm without these artifacts.

Ray-casting is the process of finding the intersection between a set of rays and object(s). This process can be used in computer graphics for high quality visualization. However, the process has been too computationally expensive to be feasible for realtime visualization of CAD-models. The recent developments in processors, both CPU and GPU, inspire researchers to develop algorithms for realtime ray-casting.

Geimer et al. [22] presented an algorithm for efficient ray-casting of trimmed bicubic Bézier surfaces. Their method exploits the vector capabilities of modern CPUs to achieve interactive frame rates. Their method takes cubic B-spline models as input and splits them into Bézier patches. Bounding box tests can thereafter be used to find potential patches for the intersection test as well as give good starting points for Newton iterations. If the expected screen size of a patch is too large the patch is split. Splitting the model into patches is performed as a pre-process which is time consuming but necessary to obtain interactiveness.

Pabs et al. [53] use a similar approach, where the intersection tests and Newton iterations are performed by the GPU. Each bounding box contain information about the parameter domain in the original surface, giving the Newton iterations good starting points in the original surface.

Reimers and Seland [55] developed a GPU-based algorithm for ray-casting algebraic surfaces of arbitrary degree. They focus on obtaining ray equations with fast evaluation and numerically stable root finding methods. The method does not require any pre-process for the implicit surface. Currently, they report interactive frame rates for degrees up to 16. Piecewise algebraic surfaces are currently not handled by this method.

Chapter 4

Numerical Simulation

Mathematical models are often used to describe the behavior of natural phenomena. This is useful both for predicting the future and to broaden our understanding of the system. Such models are often used in natural sciences, but also in social studies. Since the behavior of systems is linked with the rate of change both in time and space, the mathematical models usually include Partial Differential Equations (PDEs), which are difficult or impossible to solve analytically. However, several algorithms that solve such systems numerically can take advantage of heterogeneous architectures.

4.1 Problem Statement

Evolution of conserved quantities such as mass, momentum and energy is one of the fundamental physical principles used to build mathematical models in the natural sciences. These models are often described by hyperbolic conservation laws (and balance laws), which our research in numerical simulation focused on.

One example of hyperbolic conservation laws is the shallow-water (or Saint–Venant) equations, modeling free-surface flow over a variable bottom topography under the influence of gravity. These can be written:

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -gh\frac{\partial B}{\partial x} \\ -gh\frac{\partial B}{\partial y} \end{bmatrix}, \tag{4.1}$$

which we write on short form as

$$Q_t + F(Q)_x + G(Q)_y = H(Q, \nabla B).$$

As illustrated in Figure 4.1, $B(x, y)$ is the bottom topography, $h(x, y, t)$ is the distance from the bottom to the (wavy) surface, $w = h + B$ is the depth-averaged velocity, and g is the gravitational acceleration. Note that the variation in the bottom topography appears as a source term on the right hand side of the equation.

Equation (4.1) is based on the depth-averaged incompressible Navier–Stokes equations. Though it is based on an incompressible model, its solution has more in common with its

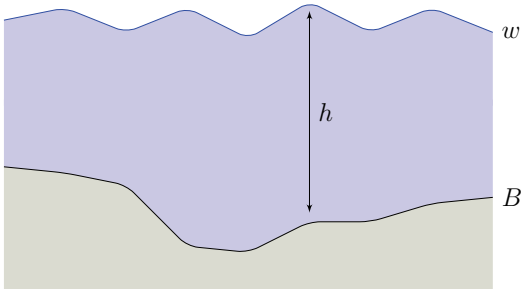


Figure 4.1: 1D view of a water surface over a varying bottom topography.

compressible counterpart due to the depth-average over a variable bottom topography. The shallow-water equations are applicable where the surface perturbation is much smaller than the typical horizontal length scale.

4.2 Numerical Methods

We superficially discuss the numerical methods here, as the focus of this document is on heterogeneous processing. For full details, please refer to Hagen et al. [26, 27].

To compute solution to Equation (4.1) it is common to use high-resolution schemes with explicit temporal discretization. Such schemes have an obvious and natural parallelism in the sense that each grid cell can be processed independently of its neighbors and are therefore ideal candidates for an implementation using stream programming model.

Equation (4.1) has two key features that make it difficult to solve numerically. First of all, the solution may contain discontinuities that correspond to breaking waves. Classical schemes, such as the Lax-Wendroff scheme [44], will therefore typically either smear the discontinuous parts or introduce spurious oscillations that pollute computed solutions. Much work has therefore been devoted to developing so-called high-resolution schemes that give high resolution in both smooth and discontinuous parts of the solution. An extensive overview of high-resolution schemes can be found in LeVeque [46].

The second difficulty with Equation (4.1) is that this system admits steady-state solutions. If no care is taken, the source terms on the right-hand side will generate nonzero numerical fluxes. These fluxes will lead to nonzero time derivatives even in the cases where the steady state is given as input. Capturing such steady-states is a challenging task for any numerical scheme. A third difficulty arises when water depths approach zero, numerical solutions may contain negative h. Negative h values are both physically incorrect and very undesirable numerically. Kurganov and Levy [42] presented central-upwind schemes that use a special strategy in areas with nearly zero water depths. In these areas, non-negative h values are guaranteed at the cost of accurately capturing steady-states.

4.3 Numerical Simulation on GPUs

Numerical simulation was one of the first topics studied in the field of GPGPU. Most of these works involve time dependent equations. The simulations start with a known state, and the simulation evolves the solution over time. Such problems are solved by an algorithm which, based on current (and previous) time step(s), calculates the solution at the next time step. Dependent on the temporal discretization of the PDE, we may get an implicit, a semi-implicit or an explicit scheme. Most GPU-based algorithms for solving PDEs are based on implicit schemes, that solve one or more linear systems for each time step. The earliest GPU-based implementations of implicit schemes were made before the programmable pipeline was available, see for example Rumpf and Strzodka [57].

Bolz et al. [7] and Krüger and Westermann [41] introduced more advanced methods that take advantage of the programmable pipeline. Historically, the GPUs lacked double precision floating-point operations. This shortcoming has been a major problem for solving systems of linear equations, especial those arising from FEM applications. Strzodka and Göddeke [61] used a mixed precision approach to obtain more accurate results whilst still taking advantage of high-performance single precision computations. Whether or not this will be profitable after the recent introduction of double precision will depend on the performance profile of the GPUs. The NVIDIA Geforce 200 series features double precision floating-point operations with only 10% of its single precision performance. Therefore, it may still be worthwhile to keep using single precision as much as possible.

Systems based on incompressible Navier–Stokes equations have got a lot of attention in the context of GPGPU. In contrast to the system described in Section 4.1, these systems are usually solved using an implicit scheme. Examples of simulations where GPU-based implementations achieved real-time performance are presented in Goodnight et al. [23] and Harris et al. [30].

Fan et al. [16] used a GPU cluster simulate the dispersion of airborne contaminants. The cluster was based on commodity hardware representative for its time. Each node had two Intel Pentium4 CPUs running at 3 GHz, and one NVIDIA 5800GX Ultra GPU. The simulator used a Lattice Boltzmann model (LBM) in three dimensions which was solved using an explicit scheme. A speedup of $5 - 7\times$ was obtained compared to their original CPU version on the same cluster. Later, Zhao et al. [66] extended the work to use locally refined LBM to obtain higher resolution in areas near obstacles or for areas of special interest. This approach can be used to obtain visually good results in the focus areas while the global solution remains plausible.

Chapter 5

Preparation of 3D Models for FEA

Finite Element Analysis (FEA) allows engineers to analyze properties like heat conductivity and strength in a simulated environment. The analysis can be performed on simple components, such as beams, or on complex models and large assemblies such as models of entire airplanes. The components usually originate from 3D scanning or CAD systems. Even though CAD systems are integrating FEA tools into their tool chains, transformin a CAD model to a fully usable FE mesh remains a manual and time consuming process. In the automotive industry it can take up to four months to create a FE mesh for a car. In this text we use the terms mesh and FE mesh for shape representations adapted for simulation purposes.

There are works in progress to allow CAD-and FEA systems to use the same geometric representation. Iso-geometric analysis is an initiative aiming at providing such representations. Until such methods become mainstream, the CAD-and FEA systems use different representations with their own strengths and weaknesses.

5.1 Problem Statement

One of the main stages in 3D model preparation is the geometric shape simplification operator. The main objective is to find a triangulation that represents the same mechanical properties without featuring shape details that are not compatible with the mechanical hypotheses and the FE mesh generation process. However, this shape simplification should have a small, or preferably no effect on the mechanical behavior modelled.

Due to the wide variety in FEA as well as in component shapes, we need a flexible framework that can handle different error estimates to guide the simplification. In addition, the framework must be able to transfer properties from the original mesh onto the simplified one in order to improve the shape simplification monitoring process. Our goal is to develop a shape-simplification framework that satisfies the aforementioned criteria and can be implemented on heterogeneous architectures scaling well with a high number of computational cores.

5.2 Preparation Process

CAD models use smooth shapes such as NURBS, sphere segments, and torus segments to represent the surface of objects. Solid objects are represented by their boundary surfaces. FE meshes

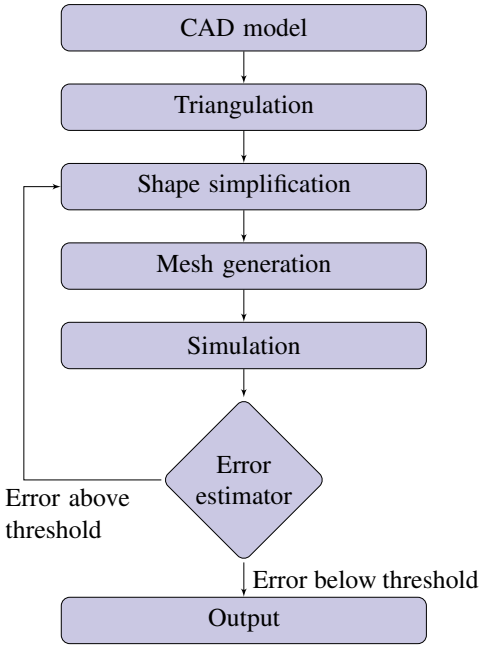


Figure 5.1: Overview of the model preparation process.

on the other hand are usually piecewise planar. Volume meshes, such as tetrahedral meshes, are often used to represent solid objects in FE analysis, and mesh generators are powerful when the input model is locally compatible with the desired FE size. Model preparation brings the CAD model to a shape that is well suited for the meshing process. Later, the prepared model is used as a starting point for meshing.

Meshing algorithms normally access surfaces through APIs, and do not fully benefit from working directly on the NURBS representation, which is complex to modify or transform. This led Owen et al. [51] to replace a CAD model by an adapted triangulation before meshing. Working on one triangulation instead of a CAD model consisting of a large number of surface patches removes the challenges related to the different parameterization of each patch. Even though the final mesh may not use triangles in the end, the flexibility of triangulations and the numerous existing algorithms makes it a feasible choice. Of particular interest is the use of shape simplification algorithms such as vertex removal operators, which form one category of shape transformation operators needed during a model preparation phase.

The domain (2D or 3D) being described by a triangulation, it is then subjected to a FE mesh generation process whose output is called the FE mesh. The target size for each finite element is dependent on the type of simulation and the simulation objectives, such as a desired accuracy. In addition, there may be a number of other important requirements for a mesh, including maximal or minimal angles, maximal valence and relative size of neighbor elements. These requirements are not meaningful in a design environment, so even if the CAD model is exported into the right format, it will not necessarily be an acceptable basis and/or shape for a FE mesh. An example of how the preparation process can be conducted is illustrated in Figure 5.1.

Computer models of product components often contains numerous details that are part of the component shape "as-manufactured". To meet the objectives and hypotheses of component behavior simulation and reduce the time spent in the FEA process, it is advantageous that these details are removed. Venkataraman et al. [64] presented an algorithm for detecting and removing chains of blends, where the radius is "small" compared to the FE size. Similar strategies can be used also for details such as small holes and fillets, bosses.

Foucault et al. [18] presented a data structure to restrict the meshing to honor the most important edges of the CAD model. They iteratively delete edges with low "importance" (angular deviation), and joins the faces previously separated by the edge. However, instead of modifying the CAD model itself a topological data structure is used on top of the CAD model, specifying the remaining edges and face groups. This allows the edges and vertices to be laid on the surface as a non-manifold scheme.

Before the simulation is performed, it may be difficult to know which parts of the model needs high accuracy and where a coarse approximation can be used. This information may only be available objectively after an initial simulation is performed, giving rise to *a posteriori* error estimators. *A posteriori* error estimators use the simulation results to determine whether or not an FE mesh is close enough to the original CAD model to act in its place for the given simulation case. However, if the correspondence between the FE mesh and the CAD model, or another reference model, is not taken care of it is difficult to use an *a posteriori* error estimator to increase the accuracy. Hamri [29] described a triangulation data structure, maintaining the connection to the original CAD model. This makes it possible to locally refine the triangulation where the *a posteriori* error estimator indicates that simulation accuracy, and hence the geometric accuracy, need to be improved.

5.3 Shape Simplification Operators

Shape simplification is important in computer graphics as well as in preparation of models for FEA. In computer graphics, it is used to reduce the number of polygons to render as well as to limit the bandwidth needed to transfer 3D models. Shape simplification is useful for a wide range of applications which led to many different approaches to solve this problem, each useful for its applications. Gotsman et al. [24, 31] give a good overview of simplification algorithms, and we will here describe some of the key features of selected algorithms and data structures.

Simplification algorithms are often categorized based on their restriction to height fields or parametric surfaces, if they are able to maintain topology, and their ability to operate on non-manifold objects. The main focus here is methods operating on manifold surface-triangulations, since this is the most common situation when CAD models are used as input. We emphasize the ability to maintain the object's topology and the possibility to extend the algorithms to also support non-manifold surfaces. In some cases non-manifold configurations can and do appear in objects used in FEA. For the non-manifold cases to be physically meaningful we must enforce consistency rules.

Most simplification algorithms are based on removing one vertex at the time, possibly grouping removals into passes. This is known as vertex decimation. Before a vertex is removed, a number of *decimation criteria* are tested. The most common decimation criteria either restrict

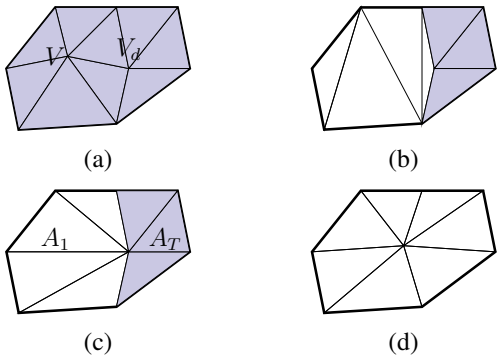


Figure 5.2: Vertex removal operators, unchanged areas are marked in blue, (a) original triangulation, (b) arbitrary re-triangulation, (c) half-edge collapse and (d) edge collapse.

the geometric deviation between the input and simplified models, or verifies that the topology of the surface is maintained. Figure 5.2 illustrates commonly used schemes for removing a single vertex. The original triangles are shown in (a), where V is the vertex to remove. General vertex removal where the neighborhood surrounding the vertex V is remeshed freely is depicted in (b). One example of such schemes is Schroeder et al. [59], where the remeshing is determined mainly by feature edges and aspect ratios. Hoppe [35] used edge collapse, where one edge is collapsed, placing its two vertices at the same position, leaving two triangles degenerated. This operation is illustrated in (d), where the edge between V and V_d is collapsed.

Kobbelt et al. [40] introduced half-edge collapse, illustrated in (c), where an edge is collapsed by moving *one* of the vertices V onto the other V_d. The main differences between edge collapse and half-edge collapse is that the latter has a smaller affected area A_1 and keeps the original vertex locations. As with edge collapse, half-edge collapse maintains the object's topology and can be extended to support non-manifold surfaces.

Edge collapse is generalized to pair contraction, allowing non-neighboring vertices to be contracted. Isolated components can be connected, which may improve the visual quality of the simplified version. The method provides no guarantee regarding the topology of the resulting triangulation. Garland and Heckbert [21] used pair contraction in what has become one of the most commonly used simplification algorithms, "Surface Simplification Using Quadric Error Metrics" also known as QEM. In their work, vertices connected by an edge and vertices with a Euclidean distance less than a given tolerance are considered pairs. A decimation cost is assigned to each pair, based on a quadric error metric. The metric is compactly represented, using 10 scalars per vertex.

Rossignac and Borrel [56] proposed to cluster nearby vertices into a *representative vertex*, and remove all degenerated edges and triangles from the simplified triangulation. The clusters are usually chosen by dividing the 3D space into a uniform grid of cells, and assigning all vertices inside a given cell to a cluster. The representative vertex of a cluster can either be one of the existing vertices or chosen as a (weighted) average of the vertices inside the cluster. Lindstrom [47] showed that QEM also could be used in a vertex clustering setting and applied this to perform out-of-core shape simplification.

Lee et al. [45] presented an algorithm called MAPS for simultaneously decimating and parameterizing a triangulation. MAPS removes vertices iteratively. In each iteration, an independent set of vertices are removed and each newly removed vertex is localized, i.e., parameterized, in the new triangulation. In order to prioritize the removal of vertices over flat regions, MAPS prefers to remove vertices with low curvature.

Cohen et al. [11] proposed to create a simplification envelope around the triangulation and test that the simplified version is inside the envelope. The envelope is constructed in such a way that it guaranties that the topology of the model remains unchanged and that the triangulation does not self-intersect.

Foucault et al. [19] proposed to base decimation criteria on properties with mechanical meaning. The general idea is to base the simplification process on properties that are known to have direct influence on the FEA process. Indeed, volume variation is closely related to mass variation and acts as a relevant criterion when dynamics takes part to the FEA process. Monitoring the position of a center of mass can also be a relevant criterion. Other examples of mechanically-based criteria include transformation of boundary conditions to modify a pressure distribution while preserving the same resulting force. The diversity of criteria is bound to mechanical parameters used as input to the FEA process. Quantities related to the solution fields of the FEA cannot be addressed by a strictly sequential FEA process, i.e. model preparation, mesh generation, solving but requires an iterative process such as illustrated in Figure 5.1. Mechanically based criteria can become time consuming compared to shape based ones. To keep the shape simplification process within an interactive timescale, it is imperative to validate the decimation criteria in parallel.

5.4 Parallel Algorithms for Mesh Simplification

Shape simplification can be performed in parallel on shared memory architectures, providing the vertices being removed at the same time are not too close to each other. A common strategy is to create a set of vertices that can be removed simultaneously where no vertex removal influence any of the others. Such independent sets can be created in parallel. Dadoun and Kirkpatric [12] and Karp and Wigderson [39] presented algorithms that find independent set in polylogarithmic time (assuming a very high number of processors).

The main challenge for parallel algorithms is to find "good" independent sets in parallel. In a sequential setting, it is common to use greedy algorithms that use a cost function to guide the selection of vertices, and a "good" independent set is one where the decimation cost is low for all the vertices. Franc and Scala [20] proposed an algorithm for finding such a set without sorting the vertices, however their method for finding the independent set is sequential.

Botsch et al. [9] proposed to improve performance of evaluation of decimation criteria by using the GPU. In their work, the decimation criterion is expressed as threshold on the distance from the decimated surface to the original. The criterion is checked by sampling a piecewise linear approximation to a signed distance field attached to the original surface, and comparing the sampled value to the tolerance. In their work, the approximation to the signed distance field is computed using a CPU-based implementation of fast marching methods. This is transferred to a 3D texture in graphics memory. Then, the triangles that are to be checked are rendered

using this texture. This is a first contribution to an efficiency increase in a decimation operator through the use of heterogeneous processor architectures.

Note that the heterogeneous computing related work mentioned here has fixed decimation criteria. Therefore, the algorithm cannot easily be extended to facilitate arbitrary criteria. Thus, new algorithms are needed to take advantage of heterogeneous computing if other geometrically or mechanically based decimation criteria are required.

5.5 Iso-Geometric Analysis

Model preparation is necessary because CAD systems and FEA systems use different shape representations. This discrepancy in representation is not only due to the difference between FE models and shape modelling principles. The disagreement is also due to the fact that the systems have been developed independently without focus to unify the shape representations. The finite element method finds the best approximation to the problem in a given solution space. The solution space is usually piecewise polynomial, with discontinuities or derivative discontinuities between the pieces. An example class of such spaces is B-spline spaces, which is often used in representation of CAD models. Hughes et al. [36] proposed to represent the CAD models in such a way that the basis functions in the shape description also could be used as basis functions for the solution space. This strategy requires a brand new toolchain for the model generation, preparation, and analysis. However, this is only a partial answer to FEA since mechanical hypotheses and behavior hypotheses often leads to locally change the shape topology and the manifold dimension. In these cases, the geometric representation of the FE model cannot be the same as the one used in the CAD model.

Existing CAD systems rely on boundary representations composed of a set of surfaces. Each surface may be represented using different surface types, including NURBS and implicit surfaces, and the surfaces may be trimmed. Furthermore, the surfaces may not match exactly, but have a distance within a given tolerance. Trimmed surfaces and imperfect matches pose challenges when the basis is also used for the solution space. It is therefore advantageous to prepare the models to avoid these problems before the models are used for analysis. Moreover, if the analysis is performed on a volume the model must be converted from a boundary representation to a volumetric representation. This conversion is non-trivial.

Iso-geometric analysis has shown very promising results for the analysis, even in the cases where the solution is expected to have less smoothness than the CAD model. There are, however, some open questions related to modeling tools and how to integrate these tools into existing CAD software, since the iso-geometric representation is not identical to existing CAD models. Until these tools are available, more traditional methods will be used, and simplification of triangulations is likely to continue to play an important role.

Part II

Our Contribution

Chapter 6

GPU-based Screen Space Tessellation

In Section 3.1 we described some of the challenges related to high-quality interactive rendering of CAD-type surfaces. Here, we present a two-pass rendering method inspired by the algorithm by Yasui and Kanai [38]. As described in Section 3.3, their method rasterizes and renders a coarse tessellation to sample a parametric surface. The sampling is uniform in the parameter domain within each patch of the tessellation. For each sample, the position and surface normal are evaluated. These values are used to compute the color of the each sample. In the second rendering pass, each sample is rendered as a point primitive using the vertex position and color from the first rendering pass. Since they use point primitives in the rendering, their method does not guarantee that all pixels covered by the surface are rendered into. Here, we will describe the method we developed in cooperation with Hagen [33], which aims to solve this problem.

The tessellation/rendering algorithm we present here targets the NVIDIA GeForce 5 series of GPUs featuring more functionality and computational power in the fragment processor compared to the vertex processor. Therefore it is important that the surface evaluation is performed in the fragment processor and not in the vertex processor, which would otherwise be the natural choice.

6.1 Overview

Our method sample the surface by rendering an initial coarse tessellation $T(u, v)$ of a parametric surface $S(u, v)$. We continue by considering the samples as vertices of triangles, as illustrated in Figure 6.2. This yields result similar to adaptive tessellation, where each patch in $T(u, v)$ is tessellated based on its size on screen. The goal is that each new triangle covers a predefined number of pixels, independent of the relative size in the viewport of the coarse triangulation. Since the samples are connected, our method does not leave gaps in the surface when the distance between neighboring samples are more than one pixel. Small triangles ensures that there will not be noticeable artifacts like popping and flickering even though all triangles are regenerated every frames.

Figure 6.1 shows a simplified view of our algorithm. To avoid artifacts near silhouette curves, the coarse triangulation is first split into charts. This pre-process is performed by the CPU in step ①. Step ② renders the charts storing the output in two separate off-screen buffers. One buffer is used to store the surface positions evaluated in the fragment shader. The other

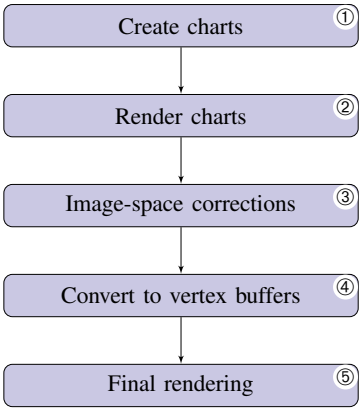
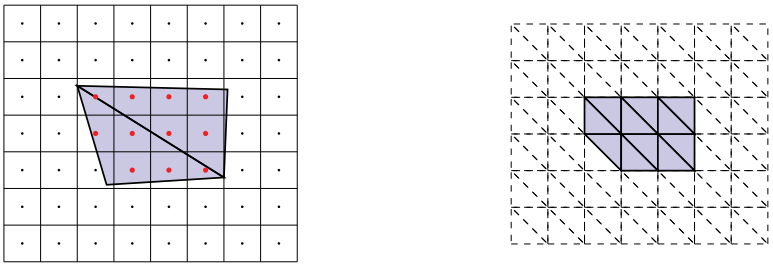


Figure 6.1: Simplified flowchart of our algorithm.



(a) (b)

Figure 6.2: Figure (a) illustrates the rendering of the coarse triangulation. The pixels marked in red are rendered into, generating valid vertices. Figure (b) illustrates the generated triangles. The dashed triangles are incident to at least one invalid vertex, and must be discarded.

buffer is used to store the associated parameter values. A point $S(u_0, v_0)$ may be within the viewport while $T(u_0, v_0)$ is outside the viewport of the off-screen buffer. Then the final rendering will not cover this part of the surface, and the boundary of the viewport will remain blank. This can be avoided by expanding the viewport in the off-screen buffer until the viewport covers the missing parts of T. By calculating an upper bound of $\|S(u, v) - T(u, v)\|$ it is possible to determine the necessary expansion. Step ③ performs modifications necessary to avoid artifacts near the boundary of the object.

Step ④ copies the two off-screen buffers into vertex buffers. The values in the vertex buffers are not treated as pixel values, but rather as one-dimensional arrays of vertex attributes. Such arrays can either be rendered directly, or using an *index array*. In our case, three consecutive elements do not form a triangle, and an index array must be used. Since the resulting vertices originate from pixels, they are arranged in a regular pattern. We therefore use triangle strips with predefined index arrays to perform the final rendering in step ⑤. Figure 6.2 illustrates how triangles are formed from the pixel buffers. Colored pixels contain vertex positions. Note that

we chose the diagonal from lower right to upper left when defining the triangles. The opposite diagonal would give similar results.

In step ⑤, the final rendering pass, the vertex buffers are used as vertex arrays specifying positions and texture coordinates. For this rendering, we use a fragment shader that computes the final color of each pixel. The triangles rendered will in general be small enough for per-vertex normal evaluation. However, in order to achieve the highest quality we used the fragment shader to evaluate surface positions and normals. The result are then used for per-pixel lighting calculations. Bumps in the surface will be captured even if the sampling in step ② is sparse compared to the frequencies in the surface geometry. The triangles marked with dashed lines in Figure 6.2, are incident to at least one invalid vertex and must be discarded. The vertex shader in step ⑤ must therefore be able to detect invalid vertices, and prohibit the triangles from being rendered. We use the alpha value in the position buffer to mark invalid vertices. In the final rendering we use a vertex shader that, based on the alpha value, detects if the vertex is valid (originates from a pixel updated by the first pass rendering). If the vertex is valid the position is transformed by the affine transformation defined by the fixed function matrices. Otherwise the position is set to $(0, 0, -\infty)$. This ensures that triangles containing invalid vertices are degenerated and not rendered. Observe that this also holds for triangles containing only one invalid vertex because the triangle becomes parallel to the viewing direction.

Before describing step ① and ③ in more detail, we give the outlines of a program performing the steps described so far:

1. Clear the position buffer such that the alpha value is set to zero.
2. Set up the graphics system for rendering to more than one off-screen buffer. Enable a fragment shader that evaluates S. The fragment shader outputs the evaluated positions into one buffer and the parameter values into the other buffer.
3. Render the initial coarse tessellation T, completing step ②.
4. Copy the off-screen buffers into vertex arrays, step ④.
5. Set up the graphics system for rendering to the frame buffer. Set up the graphics system to use the vertex arrays from step ④ such that the vertex arrays containing positions and parameter values are used as vertex position array and texture coordinate array respectively. Enable the vertex shader that detects if the vertex is valid, and sets the position of invalid vertices to $(0, 0, -\infty)$. Enable a fragment shader which evaluates the surface normal of S, and does the lighting calculation based on this normal.
6. Initiate step ⑤ by rendering a predefined index array.

6.2 Creating Charts

The triangles rendered into the final image consist of vertices that correspond to parameter values associated with triangles rendered in step ②. Near silhouette curves, this may cause parts of the surface that should have been visible to be missing from the rendered image.

Silhouette curves are important features of a surface, and defects in the visualization of these curves are easily detected by the user. A key feature of a view dependent tessellation is its ability to produce good silhouette curves. Near a silhouette curve there may be parameter values associated to back faces on T where the surface normals of S point towards the camera. These parts of the surface will be missing from the rendering, unless we modify T. Silhouette edges on T can easily be detected. Since T is expected to be a coarse tessellation, brute force methods can be applied. In order to fix the tessellation we extend the surface across such silhouettes, thus creating strips of triangles containing the parameter values of the back facing triangles.

From a given viewing direction T may consist of several layers of geometry. From such angles, parts of T will be occluded by layers closer to the viewer. This effect is enhanced by the added strip of triangles near the silhouette curves. It is important to ensure that points from different layers are not falsely connected in the new tessellation. Therefore we split T into charts in such a way that false connections do not occur. Each chart may contain points from different layers of T, therefore care must be taken so that false connections are avoided. False connections can happen when points that are not neighbors in the chart are rendered into neighboring pixels. We avoid this by requiring that each chart projected onto the viewport is not self-intersecting within an intersection tolerance of one pixel. To simplify the image-space corrections in step ③, the edges along the boundary between two charts should not be less than 45°.

To improve the performance, it is desirable to render all charts into the same off-screen buffers. We do this by estimating the size of each chart and then pack the charts into non-overlapping domains of the off-screen buffers.

6.3 Image-space Corrections

The final rendering is watertight within each rendered chart. Step ③ in our algorithm ensures that this also applies for the seems between the charts. As described in Section 1.7.1, the midpoint of a fragment is used in the linear interpolation of vertex attributes. Therefore the vertex positions calculated in step ② at fragments nearest the seem will generally not belong to the boundary of the chart. This will also cause the final rendering to be shrunk related to S.

To remedy this problem, we render the boundary edges as one-pixel wide lines. Thus, the charts at both sides of a seem will sample the border at the same boundary points. Special points along the surface boundary such as the corners of the parameter domain should be interpolated by the rendering. This point interpolation is achieved by also rendering each interpolation point as a point primitive after the edges are rendered.

Near a vertex v where the angle between the (projected) boundary edges is less than 45° the pixels may form unfortunate patterns where vertices are discarded during triangulation because all triangles they are part of are degenerated. The three pixels furthest to the right in Figure 6.3(a) form such a pattern. The pattern occurs when the edges on both sides of a v are drawn into the same pixels. We detect the areas where such patterns may occur on the CPU. By utilizing the stencil buffer we detect the pixels that are drawn into by the both edges. After detecting the pixels causing the artifacts, their values are replaced with the corresponding values for v. This result is illustrated in Figure 6.3(b). Note that a consequence of this is that the

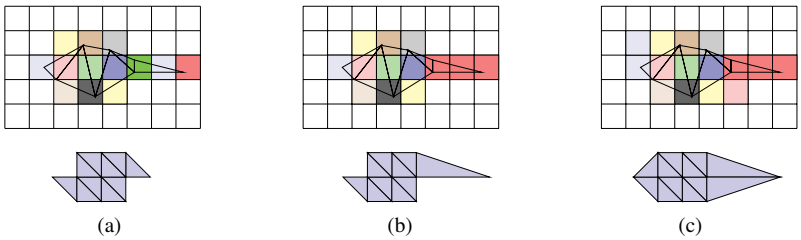


Figure 6.3: Top: shows the rasterization of the initial tessellation T, and how the off-screen buffers are modified near the boundary. Bottom: illustrates the topology of the resulting tessellation. In (a) and (b) the result before and after fixing the ignored vertices are shown. The result after the image-space corrections is illustrated in (c).

triangles incident to v are somewhat enlarged.

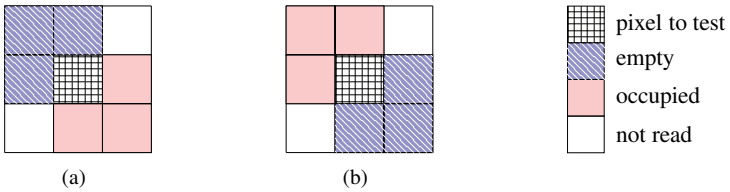


Figure 6.4: Pixel configurations that cause jagged edges. Configuration illustrated in (a) and (b) corresponds to the upper left and lower right parts of Figure 6.3(b) respectively. The result after the fix is shown in Figure 6.3(b) and Figure 6.3(c).

As mentioned, we create triangles by connecting neighboring pixels as illustrated in Figures 6.2. The choice of lower right to upper left diagonal was arbitrary, and in some configurations the choice of diagonal will make the boundary of the surface jagged, and gaps along the seems between charts. This situation is illustrated in Figure 6.3(b). One solution is to render both choices of diagonals, which would double the triangle count in step ⑤, and thus almost halving the performance. A far more efficient solution is to resolve the problem in the image space as part of step ③. The jagged edges can only occur in the cases shown in Figure 6.4. An efficient shader can detect and repair these cases, giving the result shown in Figure 6.3(c).

6.4 Hindsight

At the time we presented our GPU-based screen space tessellation algorithm, GPUs lacked much of the functionality available today. Still, the resulting view-dependent tessellations are of high quality and suitable for visual inspection of parametric surfaces. A coarse tessellation of a free form surface usually captures the main characteristics of the surface. The geometry of such a tessellation is often simple. Therefore, real-time performance is obtained using simple methods for detecting silhouette curves and splitting the initial tessellation into charts. The

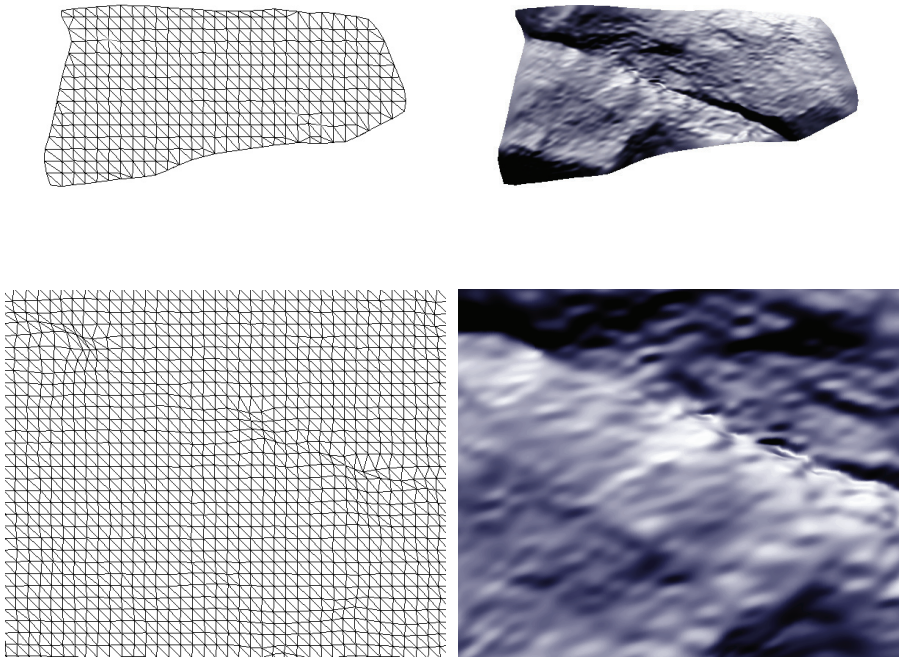


Figure 6.5: Visuzlization of a cubic spline surface. Top: the entire surface. Bottom close-ups of the center.

renderings depicted in Figure 6.5 were achieved with realtime performance using a NVIDIA GeForce 5800FX. Note that the triangle size stays the same for close-up as for the full image.

More recent GPUs can access textures from vertex shaders. This has made it possible to evaluate the surface in the vertex shader directly. Furthermore, since new GPUs use the same arithmetic units for all shader types there is no longer advantage to use the fragment processor for surface evaluation. Therefore, rendering using template geometry such as Guthe et al. [25] has become popular.

Dyken et al. [15] perform on-the-fly refinement of silhouette triangles while they are rendered. The refinement level is determined for each edge, based on its on-screen length and if it is near a silhouette. The refinement level of an edge is equal for both triangles sharing the edge, leaving the resulting mesh "watertight". In their implementation, a Bézier surface interpolating the input mesh and its normals gives the position of the new edges (and their normals). Their algorithm can be used for visualization of NURBS surfaces as well.

In the near future DirectX 11 compatible hardware featuring a tessellation step in the graphics pipeline will be introduced. This is likely to make possible simpler algorithms achieving the same visual quality as our method with even better performance.

Chapter 7

Visual Simulation of Shallow-Water Waves

The research concerning simulation has been performed as a joint work at SINTEF. Lie and Natvig shared the main responsibility for finding numerically stable algorithms that are well suited for GPU-based implementations. Hagen and Hjelmervik had the main responsibility for the GPU-implementation. The latter which is the main focus here and we describe the numerical methods only with the detail level required to present the GPU implementations. A more detailed description of the numerical methods and our implementations is presented in [27, 26], where we also presented our GPU implementations of classical schemes. The work presented is related to solving the shallow-water equations given in Equation (4.1). Recall from Section 4.1 that these can be written:

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -gh\frac{\partial B}{\partial x} \\ -gh\frac{\partial B}{\partial y} \end{bmatrix},$$

where $B(x, y)$ is the bottom topography, $h(x, y, t)$ is the distance from the bottom to the (wavy) surface, $[\, u, v \,]$ is the depth-averaged velocity, and g is the gravitational acceleration.

7.1 Overview

To solve the shallow-water equations we used high-resolution schemes with explicit temporal discretization. Such schemes have an obvious and natural parallelism in the sense that each grid cell can be processed independently of its neighbors. Therefore, these schemes are ideal candidates for implementation on heterogeneous many-core architecture such as GPUs. Here we shall compute and compare approximate solutions to Equation (4.1) using the GPU and the CPU. Both implementations use single precision floating-point arithmetic. We demonstrate that the GPU gives a speedup of $15 - 40$ times. The numerical stability of the scheme used ensures that the accuracy is sufficient even on large grid models. The speedup therefore makes the GPU an interesting and inexpensive alternative to high-performance computers for qualitative and quantitative simulations of physical phenomena. Although our main focus is on the usability and applicability of GPUs in scientific computing, we also try to demonstrate that numerical solution of the shallow-water equations can be used to create semi-realistic, nonlinear wave effects in interactive visual applications. Traditional CPU implementations do not yield

interactive frame rates for high resolution schemes of the same grid sizes.

Here, we use the short form of the shallow-water equations,

$$Q_t + F(Q)_x + G(Q)_y = H(Q, \nabla B).$$

To evolve the solution in time, we use an estimate of point samples for $F(Q)$ and $G(Q)$. The point samples are estimated by reconstructing one linear function for each grid cell, based on the neighboring cell averages. The edge fluxes $F_{i+1/2,j}$ and $G_{i,j+1/2}$ are then approximated using a standard two-point Gaussian quadrature. The edge fluxes are used to compute the time derivative of the solution using:

$$\frac{dQ_{ij}}{dt} = -\big(F_{i+1/2,j} - F_{i-1/2,j}\big) - \big(G_{i,j+1/2} - G_{i,j-1/2}\big) + S_{ij}, \tag{7.1}$$

where S_{ij} and Q_{ij} are the cell averages of the source term and the solution respectively. Equation (7.1) gives one ordinary differential equation per grid cell. These equations are solved using a second-order total variation diminishing Runge–Kutta method. A condition for convergence of the solution is that the timestep must be less than the time it takes a wave to travel one grid cell. This is usually referred to as the Courant-Friedrichs-Lewy (CFL) condition. In our case, this can be determined by finding the maximum eigenvalue of locally defined Jacobian matrices.

In Section 4.2, we introduced a scheme by Kurganov and Levy [42] that is able to capture both steady-state solutions and guarantee non-negative water depths. We adopted their strategy, reconstructing two different sets of variables depending on the water depth. To resolve the steady-states, we reconstruct point values from variable set $[w, hu, hv]$, where $w = h + B$. The exception is the areas where the water depths are close to zero, where it is more important to ensure that h is non-negative, and the variable set $[h, u, v]$ is reconstructed.

7.2 Implementation on the GPU

We have implemented the method presented above using the `Shallows` library together with the NVIDIA Cg shader language. The general setup is to render an oversized triangle that covers the entire viewport, and employ a fragment shader as the computational kernel. For a quadrilateral domain it is more natural to render a quad than a triangle. However, a quad would be converted into two triangles before rendering, which reduces the performance.

We based the implementation on the stream programming model described in Section 1.3. Thus, the algorithm must be divided into steps, in such a way that each data cell can be calculated independently within each step. Each step is executed by rendering a triangle covering the viewport—which causes the fragment processor to execute the fragment shader, thereby calculating the new cell values. Figure 7.1 illustrates the different steps and the data flow in the implemented algorithm.

Initial data are created using height maps of the bottom topography and water level. Step ① transfers the initial data to a texture in graphics memory. In steps ② and ③, the CFL stability condition is determined by the global maximum of eigenvalues in the grid cells. To find this maximum we use an 'all-reduce' operation utilizing the depth buffer combined with read-back

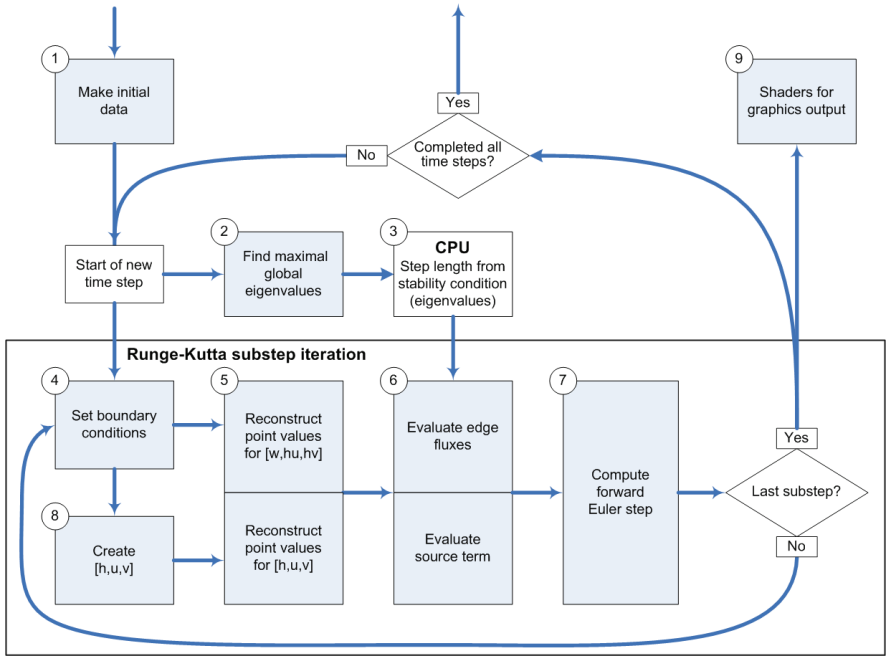


Figure 7.1: Flowchart for the GPU implementation of the semi-discrete finite-volume scheme. Gray boxes are executed on the GPU and white boxes on the CPU.

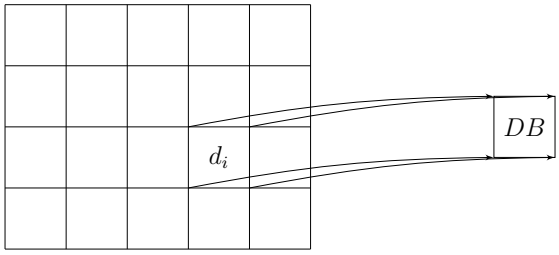


Figure 7.2: "All reduce" operation using the depth buffer. The computational domain D is divided into subdomains. Each subdomain is rendered into the depth buffer DB.

to the CPU for the final calculations. To this end, we divide the domain D into smaller subdomains d_i and render the eigenvalues of each subdomain d_i into the depth buffer. As illustrated in Figure 7.2, the depth buffer used has the same dimension as d_i. The effect of this is that one of the values of the depth buffer must be the largest eigenvalue. The depth buffer is then read back to the CPU, which picks the global maximum. We did not use the GPU to determine the global maximum because the application requires the maximum value to be read back to the CPU anyway. Compared to a linear search through all eigenvalues by the CPU, the depth buffer technique gives a performance gain, mainly due to the efficiency of the GPU's depth test. With this method, this part of the algorithm represents less than 5% of the total computational cost.

The Runge–Kutta method for advancing the solution in time comprises two iterations. Each iteration begins with step ④, a fragment shader that sets the boundary conditions. Next, step ⑤ is to reconstruct point values from the cell averages. For cases with positive water depth everywhere in the domain, it is only necessary to execute the fragment shader for the variables $[w, hu, hv]$. For computations with dry states, we also need to reconstruct point values of $[h, u, v]$. In this case, we compute $[h, u, v]$ at step ⑧, and then apply the reconstruction shader to both sets of variables.

The most computationally intensive is step ⑥, where the edge fluxes and source terms are evaluated. The time step Δt is passed to this shader by the CPU when the stability calculations of step ③ have completed. For the simplest case with constant bottom topography, $\nabla B = 0$, the source terms are zero, and the shader computes only the edge fluxes. For the more complex case of varying bottom topography, the source terms are also needed. When the simulation involves dry states, branching is needed at this step to determine if one should use the reconstruction of $[w, hu, hv]$ or $[h, u, v]$. The design of the graphics hardware is not optimal for handling branching efficiently, so generally one should aim at using algorithms with as few branches as possible. Eliminating the branch in the scheme used seems rather difficult—for cases with dry states, computational performance may therefore be better with other schemes; see e.g., Audusse et al. [3].

Step ⑦, forward Euler, evolves the iteration during each Runge–Kutta substep. By repeating step ④-⑧ one arrives at the second-order Runge–Kutta scheme. A number of different shaders are used in step ⑨ to output the result to the screen. In Figure 7.3 the water surface is rendered as a semi-transparent surface together with the bottom topography. The GPU used in this work is more advanced than the one used in the screen-space tessellation algorithm presented in Chapter 6. Of particular interest is the possibility to read textures from a vertex shader. This allows the vertex shader to read the water depth texture.

7.3 CPU versus GPU

In this section we compare CPU and GPU implementations. We measure runtimes and explore the factors that affect the relative speedup. The GPU is a NVIDIA Geforce 7800 GTX and the CPU is a 2.8 GHz Intel Xeon (EM64T).

Lake-at-rest. We consider lake-at-rest defined by a variable bottom topography $B(x, y) = \max(0, 1 - x^2 - y^2)$ and a steady state flat water surface $h(x, y, 0) = \max(w_0 - B(x, y), 0)$. Lake-at-rest is particularly interesting because numerical inaccuracies easily becomes visually

Table 7.1: Runtime in milliseconds per time step and speedup factor ν for a CPU versus a GPU implementation of the second-order central-upwind scheme. The case is 'lake-at-rest' with surface level w_0, computed on a set of uniform grids with $N \times N$ cells. For $w_0 = 1.01$ we used the simpler scheme without the switch for dry-regions.

	without dry states, $w_0 = 1.01$			with dry states, $w_0 = 0.9$		
N	CPU [ms]	GPU [ms]	ν	CPU [ms]	GPU [ms]	ν
128	32.7	1.35	24.2	35.2	2.38	14.7
256	130	4.40	29.5	143	8.09	17.7
512	518	17.2	30.1	599	31.9	18.8
1024	2140	69.8	30.6	3270	142	23.0

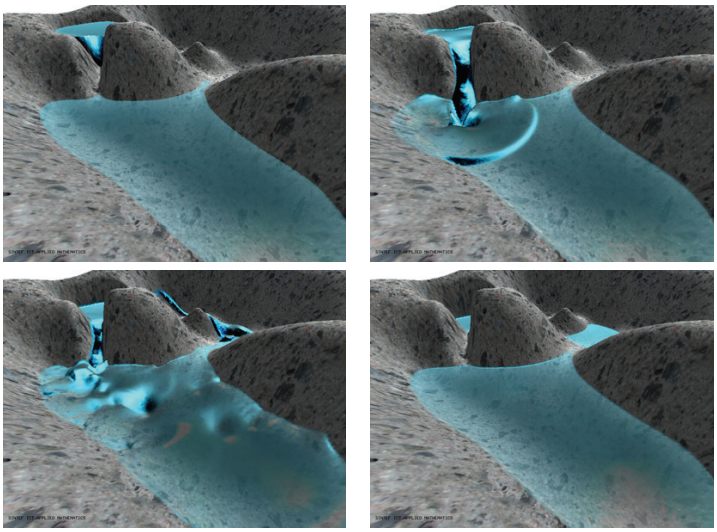


Figure 7.3: Snapshots of a dambreak simulation in artificially constructed terrain. For visual effects, the gravity is reduced in this examples.

noticeable. We were not able to visually distinguish between the results from the CPU implementation and the GPU implementation. Thus, we concluded that single-precision accuracy is sufficient in our test cases. For $w_0 = 1.01$, the water depth is strictly positive and the solution is stationary and exactly preserved by the central-upwind scheme. Runtimes per time step for the CPU and the GPU are reported in Table 7.1 along with corresponding speedup factors ν.

For $w_0 = 0.9$, we have zero water depth in parts of the domain. The measured runtimes and speedup factors are reported in Table 7.1. In this case, branching is needed to ensure nonnegative water depth. As pointed out in Section 7.2, branching is not as natural on the GPU as on the CPU and may therefore increase the runtime per time step. To avoid data-dependent branching in the reconstruction of point values, both sets of variables are reconstructed in the GPU implementation. By comparing the two simulations with different water levels, we see that without any dry-states the uniform branching yields higher speedup compared to the non-uniform case. The GPU almost doubles the runtime when the dry states are introduced. The CPU implementa-

tion, on the other hand, only slightly increases the runtime, since only the required variables are reconstructed. Though computations are relatively inexpensive, this contributes to the reduced speedup.

Flood waves caused by a dambreak. In Figure 7.3, we have included four snapshots of a dambreak simulation on the GPU. Initially, the water in the upper and lower lake is at rest. When the water in the upper lake is released through the narrow canyon, it generates flood waves and vortices in the lower lake. The wave motion and the increased water level in the lower lake makes the water flow into the valley below before it again comes to rest. The simulation is interactive in the sense that it has a fly-through mode that allows the user to inspect the solution while it is being computed.

The purpose of the simulation is to give a qualitative description of the major flood waves according to the shallow water model. If we were to use the simulation for visual purposes within computer animation, several approaches could have been taken to make the water surface look more realistic. First of all, we would have chosen a less smooth bottom topography and initial water surface. Second, one could add artificial visual water effects, see e.g., Tessendorf [62].

7.4 Hindsight

Here, we have demonstrated the applicability of the GPU as a computational resource for PDE-based simulations of gravity-driven surface waves in shallow waters. Modern numerical schemes for such models are inherently parallel in the sense that very little global communication is needed in the computational domain to advance the solution forward in time. Therefore, this application can readily exploit the parallel architecture of GPUs. We have seen in practical computations that moving from a serial CPU-based implementation to an implementation on a GPU decreases the runtime by more than one order of magnitude. For instance, the runtime of the full dambreak simulation was reduced from two hours to five minutes! To achieve the same speedup using CPUs, one would have to resort to a cluster of twenty or more processing nodes.

In our experience, numerical solution of conservation laws and balance laws are as reliable on the GPU as on the CPU; the single-precision arithmetic does not negatively affect the computations. Indeed it should not, since the methods we have examined are numerically stable. The high speedup we have seen is due to the ratio between arithmetic operations and memory access is well balanced. This allows the application to take advantage of both the memory bandwidth and the computational power of the GPU.

Data dependent branching on GPUs is expensive. There exist several known methods to conquer this challenge, but the results are highly dependent on the input data and the graphics hardware. Therefore we did not invest a large amount of time to find the best possible solution for our hardware, but aimed for a reasonable compromise. Due to the architecture of the GPUs available during our research, fragments that are being evaluated simultaneously must execute the same branch. Thus, the evaluation of some fragments must wait for its neighbors. In our application, however, the effect of this is small because the outcome of the conditionals are mainly the same for fragments closely related in screen space. Since we published the paper on this topic, new generations of GPUs have been released where the cost for data dependent branching has been reduced. However, the cost it still much higher than on CPUs.

Based on the work presented here, Hagen et al. [28] developed a GPU-based implementation solving the 3D Euler equations for the dynamics of an ideal gas. In their work the 3-dimensional domain was laid out as 2D slices, using 2D textures and frame buffers. Their work also discusses how to extend the solver to use a multi-GPU system.

As described in Section 1.7.3, GPUs now feature random access write operations, and a shared memory for GPGPU computations. In our case, shared memory can be used to reduce the number of memory reads, and improve the performance in the "all-reduce" operation. Furthermore, using a GPGPU API omitting the graphical API can reduce the overhead related to initiating computations. Recently, Michalakes and Vachharajani [50] implemented a module of "The Weather Research and Forecasting Model" system [17] to use GPUs. The most computationally intensive part of the module was written as CUDA kernels. This resulted in a $5 - 20\times$ speedup of the module, yielding an overall speedup of 25%. This is an impressive result demonstrating that porting small portions of code to use GPUs can have a noticeable effect on the runtime of larger systems.

Brandvik and Pullan [10] used the shared memory when they developed a CUDA version of a three-dimensional solver for Euler flow with boundaries. In their work the domain is split into sub-blocs that fit in the shared memory. This drastically reduces the workload for the memory bus, and therefore yields a performance improvement.

Chapter 8

GPU-Accelerated 3D Model Preparation

The problem statement presented in Section 5.1 describes the need for faster 3D model preparation algorithms. More specifically, faster shape simplification algorithms taking advantage of heterogeneous computing.

To meet this need, we developed two different algorithms. In this chapter, we presents a hybrid GPU-CPU implementation, where the CPU is used to hold the triangulation data structure and the GPU accelerates computations. This algorithm was first presented in Hjelmervik and Léon [32]. Our algorithm guarantees that the geometric error is within a given tolerance, as well as providing support for mechanically based simplification criteria.

8.1 Overview of the Algorithm

Our algorithm targets DirectX 9 generation GPUs, lacking random access write operations from the fragment processor. Therefore, it is not feasible to use the fragment processor to update triangulation data structures. However, the GPU has much higher floating-point capacity than the CPU, we therefore off-load as much as possible of the floating-point operations to the GPU. This strategy allows us to use a standard data structure for the triangulation while taking advantage of the computational power of the GPU. We found that the most computationally intensive tasks are related to validation of the decimation criteria, and remeshing the boundary edge loop of the removed faces. This is especially true when mechanically-based criteria are considered in addition to shape-based ones. Therefore, we move these computations to the GPU to reduce the computation time.

The flowchart given in Figure 8.1 describes one pass of our algorithm. Similarly to MAPS by Lee et al. [45], an independent set of vertices, IS_v is considered for removal per simplification pass. However, with the range of applications considered here, the vertex removal process terminates either under distance-based criteria or mechanically-based ones rather than a targeted number of faces. This principle means that locally very coarse meshes can be produced while staying compatible with the mechanical requirements, e.g. local size of the FE required in the FE mesh generated after shape simplification process. At each simplification pass, the vertices that potentially can be removed are sorted based on discrete curvature values. Then, a greedy approach is used to create an independent set of vertices. An example of IS_v is illustrated in Figure 8.3.

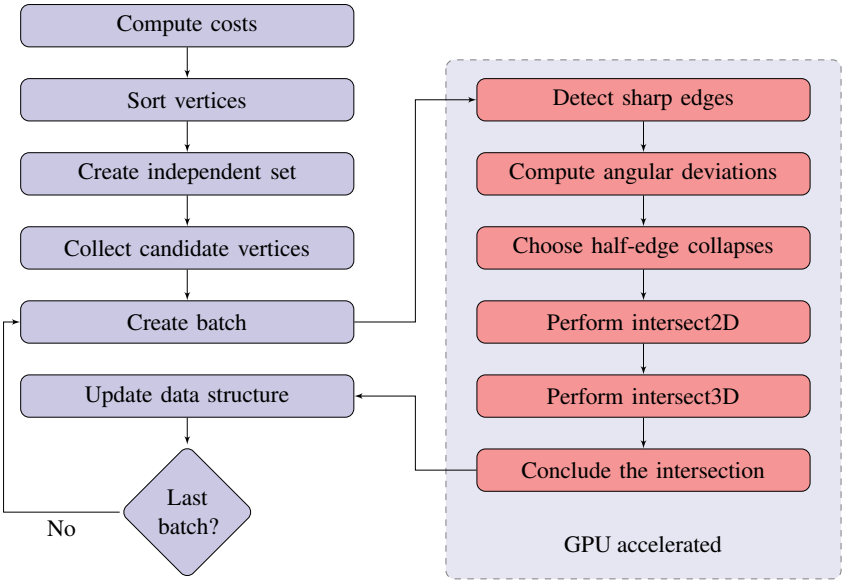


Figure 8.1: Flowchart for the GPU accelerated simplification algorithm.

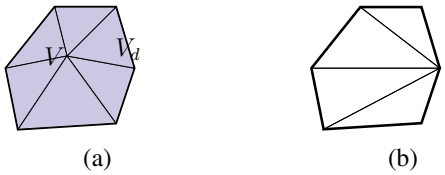


Figure 8.2: Half-edge collapse, V is collapsed into V_d.

The vertices in IS_v are then candidates for removal, and will be removed if they meet the decimation criteria. The use of an independent set ensures that the removal of any vertex in the set does not influence the removal of other vertices in the same set. However, it restricts the decimation criteria to use information inside the one-ring neighborhood only, since data located outside this area may be modified during the same pass. Allowing the decimation criteria to use information outside this region would reduce the number of vertices that can safely be removed within one pass. Due to the overhead related to initiating rendering, it is advantageous to keep the number of simplification iterations low.

Geometrical information about the candidate vertices and their one-ring neighbors are collected and transferred to the GPU. Since this information may not fit in a texture due to hardware constraints, we split the candidate vertices into batches. Each batch is treated at the GPU, where the remeshing is computed and the criteria are evaluated. Based on the computations performed on the GPU, the CPU removes a subset of the candidate vertices from the data structure of the triangulation.

8.2 Remeshing Scheme

The GPUs have little support for complex data structures, and therefore our algorithm is based on half-edge collapse to reduce the complexity of the remeshing scheme. Figure 8.2 illustrates a half-edge collapse, where one edge connected to V is collapsed, i.e., the vertex V is moved into V_D, as described in Section 5.3. Such a choice is also compatible with FEA-based applications. Indeed, for most of the simplification passes, the restriction in choice of remeshing imposed by the half-edge collapse does not affect the quality of the simplified model.

During the last stages of iteration, accessing the widest possible range of remeshing schemes is critical to widen the diversity of shapes that can be reached at the end of this process. The use of more general remeshing schemes at the end of the process is important and has a significant effect on the overall shape transformation process. General remeshing schemes can avoid too early termination while satisfying the distance constraints between the initial and simplified shapes as well as other mechanically-based criteria. Due to hardware limitations of the GPU, only half-edge collapse is accelerated at the GPU. Since general remeshing schemes only apply to a small subset of vertex removals, the time consumption is relatively small compared to the entire simplification process. Similarly, the half-edge collapse scheme addresses configurations equivalent to two-manifold ones. In the present contribution, non-manifold and surface boundary configurations are still handled by the CPU since they occur far less frequently than two-manifold ones.

When a candidate vertex is considered for removal using half-edge collapse, one has the freedom to choose destination vertex. Our choice is based on properties relevant in a mechanical setting. Sharp edges are important properties of mechanical models and we therefore aim at keeping such edges in the approximated version. Therefore, we use the dot product between the normals of the two triangles at each side of an edge as the *master value*. The computations described in this section are performed for the four edges with smallest master value. This will guide the remeshing process to the sharp edges. Furthermore, compared to performing the computations for all the edges, it saves considerable amount of computation time when considering a candidate vertex of high valence.

A half-edge collapse can result in a topologically illegal configuration, such as two faces being mapped on top of each other. CPU-based implementations can check for such configurations, and change the remeshing scheme of the candidate vertex. This is not feasible in our GPU-implementation, because such tests require topological information outside the one-ring neighborhood of the candidate vertex. We therefore use geometric properties within the one-ring neighborhood to detect if it is likely that a double face occurs. The angle between the new and old triangle connected to each boundary edge is used to detect illegal configurations. If this angle is larger than a predefined tolerance for any of the edges, the remeshing is likely to be topologically illegal or geometrically unwanted, and is therefore marked as unwanted. We found that an angular tolerance of 80 degrees detects almost all illegal situations, without preventing important vertex removals. This test does not replace a topology test performed at a later stage.

During a half-edge collapse, one can easily compute geometrical properties like the local volume variation. The volume variation of the prepared model relative to the original is an important property of simplified models used to generate a FE mesh. It is one of the *a priori*

objective criteria that we have addressed. It is an objective criterion because it can be quantified before the FEA takes place and it conveys mechanical meaning because it is directly linked to the mass of the object. The mass is an important mechanical property when dynamic behavior simulation is addressed. Similarly, volume variation is linked to variations in the centre of gravity and hence it is effectively at the basis of several *a priori* mechanically-based criteria. Not only global volume variations, but also local ones are important to characterize and offer to the mechanical engineer. This allows the engineer to prescribe volume variations over a subdomain of the object according to his/her mechanical hypotheses. Local evaluation of volume variation can be used also to display its distribution over the simplified shape, thus providing qualitative information regarding the distribution of the volume variation. The local volume change is therefore used to choose among the edges selected with the smallest master value (angle between the connected faces).

8.3 Decimation Criteria

A vertex is only removed if the half-edge collapse fulfils the decimation criteria described here. The two-sided Hausdorff distance yields an accurate estimate of the geometrical simplification error. However, it is computationally intensive, and it is difficult to include this criterion in interactive applications where large models are used. A less computational intensive, but accurate criterion, is the one-sided Hausdorff distance from the original vertices to the decimated mesh. This is a common choice for simplification algorithms. It corresponds to assigning an error zone to each original vertex and testing if the simplified version intersects all error zones. Véron and Léon [65] used this strategy. Each error zone can be represented as a sphere with radius equal to the user-specified geometrical tolerance. However, error zones can also take mechanical criteria into account, by appropriately adjusting the radius of each sphere. To this end, the radii can be set equal to the FE map of sizes desired, enabling *a priori* information about the FEA to be used in the simplification process. In this case, the FE map of sizes is based on the user's know-how about the mechanical behavior of the structure and his/her ability to locate stress gradients. Further, the sphere radii can be set automatically based on *a posteriori* FEA criteria, as illustrated in Figure 5.1. In this case, the sizes of the error zones reflect the sizes of the finite elements required to match the analysis accuracy specified by the user. The sphere sizes can also be defined using a strain energy error estimator based on a previous analysis to provide a new model for better FEA results. Here, the FE map of sizes is considered to be set up *a priori* only.

Each error zone is associated with one vertex and does not become active until the associated vertex is removed. The decimation criterion is satisfied if each active error zone intersects at least one triangle. The error zone is then assigned to one of the intersecting triangles. Note that the number of active error zones does not exceed the number of original vertices. This guarantees that the number of intersection tests to be performed at each simplification pass does not increase during the simplification process.

The use of error zones to incorporate *a priori* mechanical criteria allows the decimation process to take into account some of the aspects of FEA. However, it does not give the user objective information on how well the mechanical properties of the simplified version correspond

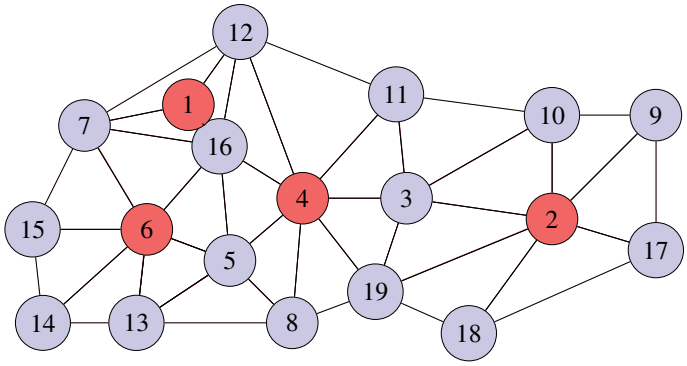


Figure 8.3: Example triangulation, independent set of vertices are marked red.

to the original model. To incorporate this kind of information, we keep track of the global variation of volume. Since the local volume variation is already computed when choosing which half-edge to collapse, it is only a matter of adding these values, and compare with the user defined global volume tolerance.

The decimation criteria performed by the GPU can only use information in the one-ring neighborhood of the candidate vertex. Any global criterion, such as global variation of volume, must therefore be evaluated at the CPU using the volume calculations from the GPU. Further, topological criteria must be performed to ensure that the topology of the object is maintained. If a half-edge collapse is found to be topologically illegal, it is omitted. Otherwise, if the candidate vertex has passed all tests described here, it is removed from the triangulation.

8.4 Data Structures

We use a standard C++ data structure for maintaining the triangulation in system memory. This data structure is used to retrieve the necessary information of all candidate vertices, which is then sent to the graphics memory as textures.

The two main textures are called `positionTex`, and `neighborsTex`, which contain the positions of the candidate vertices and their neighbors. Kernels used to determine the remeshing scheme and evaluate decimation criteria loop over the neighboring vertices. To ensure that vertices treated simultaneously spend the same number of iterations in these loops, the candidate vertices are grouped according to their valence. All vertices in the same row of `positionTex` are of the same valence. This simplifies the implementation and improves performance. Most of the computations are performed by rendering into a two dimensional frame buffer of the same dimensions as `positionTex`. Each pixel in the frame buffer represents one vertex removal.

Example: Figure 8.3 illustrate the triangulation we will use in this example. The connectivity of the triangulation is represented as an adjacency list. For simplicity, only a subset of the vertices is presented here, and only the indices are given in Table 8.1. Note that the vertices are already sorted according to their discrete Gaussian curvature. The greedy algorithms inserts the vertices 1, 2, 4 and 6 into IS_v. These vertices are therefore the candidates for removal. The

Table 8.1: Adjacency list of the first 6 vertices of the example triangulation.

Index	Valence	Discrete curvature	Neighbors
1	3	0.1	7, 16, 8
2	6	0.11	10, 3, 19, 18, 17, 9
3	5	0.12	10, 11, 4, 19, 2
4	7	0.13	11, 12, 16, 5, 8, 19, 3
5	5	0.15	8, 4, 16, 6, 13
6	6	0.3	13, 5, 16, 7, 15, 14

Table 8.2: Indices to vertices in `positionTex`.

valence 7:	4	
valence 6:	2	6
valence 3:	1	

candidate vertices are then grouped according to their valence, before the textures are created. The first row contains the vertex 1, which is the only candidate vertex of valence 3. The second row contains the vertices of valence 6, namely vertices 2 and 6, and the last row only contains vertex 4. Note that in the tables the indices of the vertices are given instead of the positions.

In `neighborsTex`, the positions of the vertices adjacent to the candidate vertices are stored. Each row contains the neighbors to the candidate vertices stored at the corresponding row in the position texture. Table 8.2 and Table 8.3 show the contents of `positionTex` and `neighborsTex` respectively. Blank cells indicate texels that are not read.

8.5 GPU-Implementation of Decimation Criteria

A kernel computing the master values described in Section 8.2 is executed for all candidate vertices with valence at least equal to four. The four edges with lowest master value are candidates for half-edge collapse and their indices are returned from the kernel. Since we need to compute the volume variation for each of the potential edges, we perform the following computations in a frame buffer where each pixel corresponds to a potential edge. To simplify the implementation, we first rotate the list of neighbors, such that the destination vertex in the half-edge collapse is placed first of the neighboring vertices. To illustrate the rotation, Table 8.4 shows `neighborsTex` after the reordering. In this example vertex 1 is collapsed into vertex 7, 2 into 3, 6 into 16 and 4 into 11. The first element of the first row is the destination vertex for the half-edge collapse for vertex 1. Therefore, the neighbor vertices to vertex 1 are rotated one element to the left.

With these preparations, the local volume variation associated with each potential edge can

Table 8.3: Indices to vertices in `neighborsTex`.

valence 7:	11	12	16	5	8	19	3					
valence 6:	10	3	19	18	17	9	13	5	16	7	15	14
valence 3:	7	16	12									

Table 8.4: Indices to vertices in `neighborsTex` after reordering to produce reference configurations.

valence 7:	11	12	16	5	8	19	3					
valence 6:	3	19	18	17	9	10	16	7	15	14	13	5
valence 3:	7	16	12									

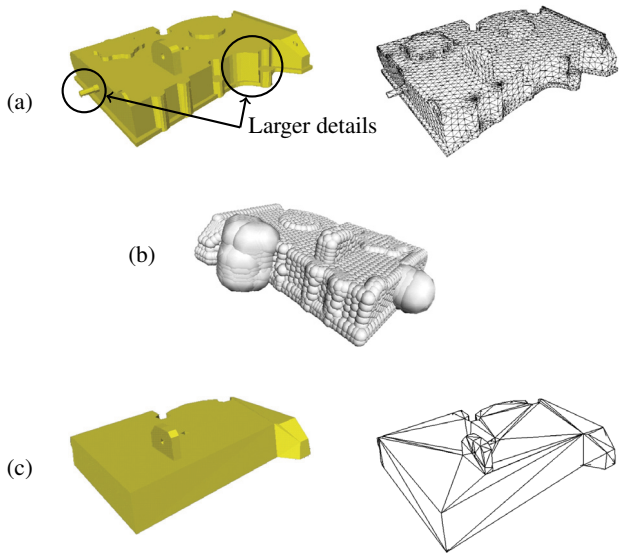


Figure 8.4: Example highlighting the effect of variable deviation distance on the shape simplification process.

efficiently be computed in a fragment shader. The only remaining operation before choosing which edges to collapse is to compute the angles between the old and new triangles, to detect flipped triangles. When the edges to collapse are chosen, we finally rotate `neighborsTex`, according to the chosen edges.

We focused on the decimation criterion based on error zones, because this allows the use of both *a priori* and *a posteriori* criteria. During the first simplification pass, i.e., before any vertex is removed, there is only one error zone per candidate vertex. This error zone is then attached to one of the new triangles. During later simplification passes, error zones originating from already removed vertices must be close enough to the candidate vertex.

To this end, intersection tests are performed by executing two kernels called `intersect2D` and `intersect3D`. First, `intersect2D` performs a simple bounding box test. Then, the kernel `intersect3D` performs the intersection test for the triangle-sphere pairs that passes the bounding box test. The `intersect2D` kernel is executed for each triangle and loops over the spheres assigned to the candidate vertex. The return value is an integer value, where each bit represents an intersection test and is set to one if the box test is successful. Since DirectX 9 does not support integer data types or bitwise operations, this is implemented using floating-point arithmetic. Then, `intersect3D` uses the bit pattern when looping over the spheres that

Table 8.5: Runtimes for simplification of mechanical models. The error zones are given relative to the length of the diagonal of the bounding box, and the volume is relative to the bounding box.

Model	Initial #tris	Error zone	Volume tolerance	CPU time (s)	GPU time (s)
Blade	1765388	1.3%	1.7e-6%	71.4	9.7
Blade	1765388	1.3%	1.7e-5%	87.6	10.8
Fandisk	12946	1.3%	1.5e1%	0.583	0.21
Fandisk	12946	1.3%	1.5e-3%	0.57	0.21

potentially intersect the triangle. A large difference in the number of associated error zones will lead to poor performance, due to the synchronized way GPUs operate. Data dependant loops is a special case of branching, and face the same performance issues as described in Section 1.7.2. Therefore, candidate vertices with a high number of associated error zones are repeated in `positionTex`, with different error zones.

Finally, a kernel called `concludeKernel` checks the results from the intersection tests, verifying that each sphere intersects at least one triangle. Otherwise, the candidate vertex is marked as non-removable. The output from this kernel is the index to the destination vertex of the edge collapse, and the volume of the approximation error associated with this half-edge collapse. If the candidate vertex is marked as non-removable, a negative index is returned.

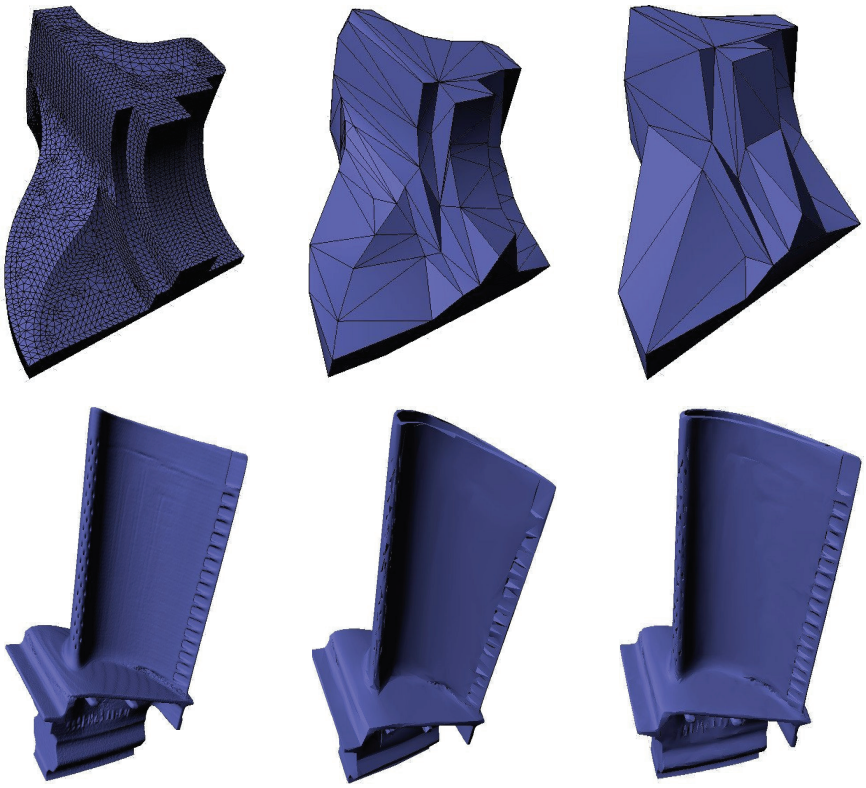
The output from `concludeKernel` and `intersect3D` are read back to system memory for the final processing. The decimation criteria performed by the GPU are restricted to be dependent on information in the one-ring neighborhood of the candidate vertex. Any global criterion, such as total variation of volume, must therefore be evaluated at this stage. Further, tests are performed to ensure that the topology of the model is maintained. Finally, the triangulation is updated and the assigned error zones are reassigned to new faces according to the results from `intersect3D`.

8.6 Results

In Figure 8.4, an example is given to highlight the influence of the error zones on the shape simplification process. In (a) the initial model shaded and tessellated, (b) is the map of sizes used for the simplification process, and (c) is the resulting model shaded and tessellated where larger details have been removed. The variable size of error zones interactively specified by the user helps identify specific subdomains as details that will be removed. Such a variable envelope adds further flexibility to meet the user's requirements.

Figure 8.5 shows how changing the tolerance for global volume variation influences the simplified model. The examples presented here are decimated with our GPU-accelerated method.

The performance advantages of using the GPU are illustrated in Table 8.5, which shows that our GPU version is 8 times faster than our CPU version for large models. The performance improvement with our GPU version is reduced for smaller models, but the algorithm outperforms the CPU version even for a model with only 12000 triangles. The models were processed using an AMD Athlon 4400+ CPU and a NVIDIA 7800GT GPU.



(a) (b) (c)

Figure 8.5: Top: fandisk model, bottom: blade model. Original models (a), simplified with global volume criterion (b) and without (c).

8.7 Hindsight

We have presented a hybrid GPU-CPU approach for shape simplification of objects dedicated to mechanical applications. The computational power of the GPUs makes it feasible to include both geometrical and mechanical criteria. These criteria are too computationally expensive for pure CPU implementations to preserve user interactivity. Since the candidate vertices are split into batches, the independent set of vertices must be created by the CPU. This prevents us from computing independent sets on the GPU without extra data transfer between graphics and system memory.

The proposed approach has highlighted the efficiency of GPU architecture where significant speed-up factors can be reached. Transferring data between system memory and the GPU can be a bottleneck if the decimation criteria is not computationally intensive.

The overhead related to data transfer and initiating computations on the GPU becomes important if too few vertices are treated per simplification pass. Using our algorithm, each simplification pass attempts to remove all vertices in an independent set. This is as good as we can expect, since the vertex removal operation takes place after all the candidate vertices are considered for removal. The half-edge collapse as the vertex removal operator provides a template remeshing scheme, avoiding data dependant branching that would otherwise lead to reduced performance. To further reduce branching, the vertices are grouped based on their valence.

DeCoro and Tatarchuk [13] presented the first algorithm to perform the entire simplification on the GPU. They implemented vertex clustering using the geometry shader to discard degenerated triangles. Since they used GPUs supporting *stream out* of vertex data, the simplified version can be reused for further simplification.

The current implementation is based on accessing the GPU through a graphics API. We expect a performance improvement if the implementation is ported to an API that allows the GPU to be used without going through the graphical system. Such APIs can increase the performance of the application and reduce the development time, making approaches like the one presented here more attractive in commercial systems.

The GPUs available today have increased flexibility and reduced cost of data-dependent branching. Revising the algorithm taking advantage of these possibilities will most likely result in significantly faster implementations. Now, it may also be feasible to also use the GPU for maintaining the data structure.

Chapter 9

Simplification of FEM-models on Cell BE

The hybrid GPU-CPU simplification algorithm we presented in Chapter 8 performs well provided the decimation criterion is computationally intensive. However, it is not suitable for simpler decimation criteria. Here, we present a simplification algorithm and its implementation on multi-core CPUs, and the Cell BE. The goal of the algorithm is to yield high scalability beyond the eight cores we tested it for on the Cell BE.

9.1 Description of the Algorithm

Our main goal is to develop a flexible algorithm for shape simplification that is suitable for many-core heterogeneous architectures. To take advantage of the high level of parallelism, it is important that a high number of vertices are removed simultaneously and that all steps in the algorithm can be performed in a parallel fashion. Existing thin cores do not have the capability to allocate or deallocate system memory, prohibiting implementation of algorithms that use dynamic memory.

The demand for performance improvement seems most crucial either when considering simplification of large models where computationally demanding decimation criteria are in play and/or when the simplification must be performed at an interactive rate. However, many applications use QEM or other computationally inexpensive decimation criteria. Performance may also be important for these applications, either to improve the interactivity or to simplify assemblies consisting of a large number of less detailed objects. Thus, low overhead is a key feature of our algorithm. Furthermore, the algorithm should be extendable to transfer properties attached to the faces (or edges or vertices) to enable the implementation of mechanical criteria like the transformation of pressure fields.

A simplified view of popular simplification algorithms is illustrated in Figure 9.1(a). Typically, an independent set of vertices, IS_v is created using a greedy, sequential algorithm, which inserts vertices one by one into IS_v. Such a set is used to ensure that no neighboring vertices are removed simultaneously. Indeed, neighboring vertices could be removed in the same pass as long as it does not occur at the same time. Since neighboring vertices may be candidates for removal in different threads, it becomes necessary to implement vertex removal in a thread safe

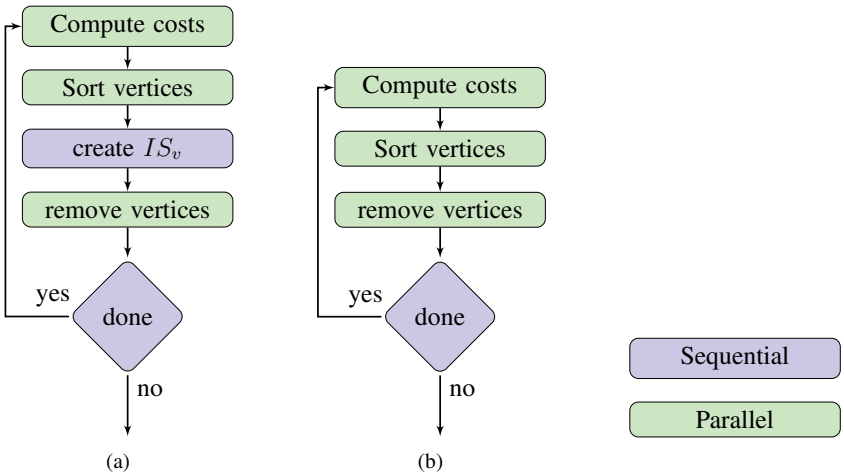
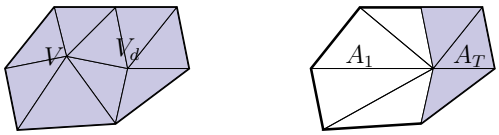


Figure 9.1: Flowcharts of simplification algorithms. Figure (a) illustrates a traditional simplification algorithm. By removing the vertices as they are inserted into IS_v we get the flowchart illustrated in (b).

manner. This is the approach we implemented and figure 9.1(b) illustrates our algorithm. This is an alternative to implementing a parallel algorithm for creating IS_v. Here, thread safe vertex removal means that multiple threads may perform vertex removals on the same triangulation, while maintaining the integrity of the data structure. If competing threads perform conflicting vertex removals, this situation must be detected and resolved. In our implementation, threads detecting a conflict will ignore the current vertex removal and continue with another candidate vertex.

Serial simplification algorithms can insert the vertices into a priority queue, and iteratively remove the vertex with lowest decimation cost. The decimation cost can be updated as the neighborhood is modified, ensuring that the priority queue is up to date. In a parallel setting, the use of a priority queue would be a bottleneck, since any update would require exclusive access. Algorithms based on IS_v are guided by the decimation cost, but as a worst case, the vertex with highest decimation cost can enter IS_v in the first iteration of simplifications. Requiring the candidate vertices to have a decimation cost less than a given threshold can improve this situation. In our algorithm, each thread performs the vertex removals in the order given by the decimation cost. However, the vertices are not reordered within one simplification pass. Depending on the objective of the decimation process, a well suited decimation criterion can prevent unwanted vertex removals, making the application less dependent of the decimation order and on the concept of threshold. Again, this underlines that it should not be required that the decimation order strictly follows the order given by the decimation cost.

Vertices not removed in the first simplification pass may be removed at a later pass. We use a *removable flag* assigned to each vertex, indicating the status of the vertex removal. Vertices failing the simplification criteria are marked as "non-removable". This is not a permanent status, since further simplifications can change their neighborhood, hence their cost and status. However, a vertex failing the simplifications criteria once is less likely to be removed than vertices



(a) (b)

Figure 9.2: Half-edge collapse, V is collapsed into V_d. To the left: the result after the half-edge collapse, The white area (A_1) is modified by the half-edge collapse. The area marked in blue, A_T is unmodified.

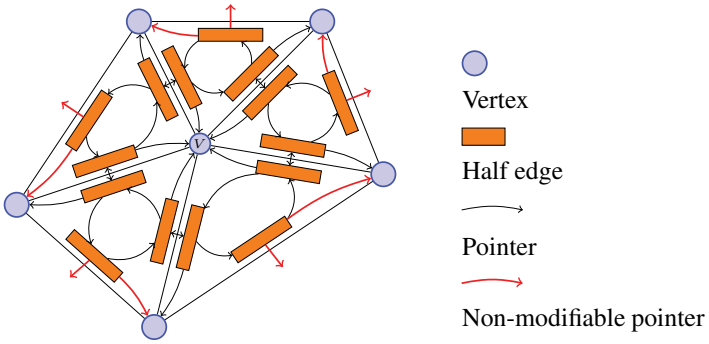


Figure 9.3: Illustration of the half-edge data structure. Pointers belonging to boundary edges and therefore not modifiable are marked in red.

not yet considered. As a compromise, we reconsider all vertices marked as "non-removable" every three pass. To restrict the SPEs to only consider the removable vertices as candidate vertices, we assign removed and non-removable vertices cost of -2 and -1 respectively. After the vertices are sorted in ascending order, the removable vertices are located at the top of the array.

A wide variety of data structures can be used to describe and store triangulations. Due to its simplicity and efficiency, vertex incidence lists are often used for simplifications. In its simplest form, each vertex contains a list of pointers (or indices) to its neighboring vertices. A vertex removal operation performed on this data structure modifies not only the removed vertex V, but also to the vertices in its one-ring neighborhood, because the adjacency relations must be updated. Thus, no other vertex in its two-ring neighborhood can be removed at the same time. Therefore one can expect more frequent conflicts than if only vertices in the one-ring neighborhood were modified.

Another popular data structure for triangulations is based on half-edges. Figure 9.3 illustrates the data elements involved in a vertex removal. The edges surrounding the one-ring neighborhood of a vertex V stay unchanged during its removal. Therefore, we only read and modify the half-edges on the "inside" of this area during a vertex removal. These half-edges are only relevant for vertex removals of any vertex inside this one-ring neighborhood. Therefore, the half-edge data structure does not increase the radius of influence beyond where the triangles change.

Vertex removal operators can result in locally topologically illegal configurations, such as

Listing 9.1: Thread safe vertex removal

```
Lock the potential vertex V
Check for conflicts
  Verify that the half-edges ''return'' to the center vertex V
  Verify that none of the neighboring vertices are locked
Check geometrical (and mechanical) criteria
Check topological criteria
Update data structure
  Update pointer to half-edges
  Update the pointers to the vertex V
unlock the potential vertex V
```

two faces being mapped on top of each other. Before removing V it must be verified that no edge is added between already connected vertices. To perform this test, it may be required to expanded the area of interest from the one-ring neighborhood of V to the two-ring. The implementation of the topological test is dependant of the remeshing scheme used. For the half-edge collapse the area of interest is expanded from the one-ring neighborhood of V to also include the one-ring neighborhood of the destination vertex V_d. In Figure 9.2 (b) the original area of interest is labeled A_1 and the added area labeled A_T. Due to the small area of interest and simple implementation, half-edge collapse is used in this work. Other remeshing schemes can be used to provide a larger range of shapes that can be reached.

A thread safe vertex removal operation must detect if either the vertex itself or one of its neighbors is being removed by another thread. Thread safe locking mechanisms are available for most parallel architectures, and we added a lock for each vertex to the data structure to implement such a mechanism. As shown in Listing 9.1, we lock the potential vertex V as the first step of a vertex removal. During the update of the data structure, we update the pointers to V as the last step, to ensure that competing vertex removal threads see that V is locked. During a vertex removal with the half-edge collapse operator, the data structure is locally inconsistent within the one-ring neighborhood of V. The inconsistencies take form either as three consecutive half-edges not forming a loop, or half-edges pointing at the incorrect vertex, i.e. the pointer to the half-edges are updated but not the pointer to the vertex. These inconsistencies can easily be detected, by verifying that there is no vertex locked and that three consecutive vertices form a triangle. If neither inconsistency or locked vertices are found, the vertex removal process can safely continue.

9.2 Cell BE Implementation

From a programming point of view, memory management related issues constitute the main differences between traditional homogeneous architectures and heterogeneous architectures such as GPUs and the Cell BE. So far, we have described vertex removal operations using shared memory architectures with coherent caches. Traditional homogeneous multi-processor and multi-core systems fit this description. However, heterogeneous architectures such as GPUs and the Cell BE are not equipped with coherent caches. Therefore, algorithms implemented on such architectures should not rely on memory transfers being performed in the order they are issued. In this section we present our Cell BE implementation of thread safe vertex removal.

Section 1.6 gives a description of the Cell BE processor, and the most relevant information

Listing 9.2: Simplified view of thread safe vertex removal on Cell BE

```
Lock the potential vertex V
While looping around the vertex V to collect neighbors
  Verify that the half-edges ''return'' to the center vertex V
  Verify that none of the neighboring vertices are locked
Check geometrical (and mechanical) criteria
Check topological criteria
Update data structure
Update the status of V
```

related to data transfers. In our work, we use a software-based cache for reading half-edges and vertices, and perform the corresponding write operations without using the software-based cache. Our algorithm uses the SPE's atomic unit for maintaining the status of a vertex. A vertex can have the following states: locked, removed, non-removable, or removable, covering the needs for both the current decimation pass and the process between the passes.

The basic data element in a SPE is a quadword, and data transfers of naturally aligned quadwords are performed as atomic operations. Based on the concept of atomic operations, we therefore store vertices and half-edges in quadwords to guarantee that we never read a partly updated vertex or half-edge. An half-edge is represented as three indices (four if face attributes are explicitly represented), while a vertex is represented as three floating-point numbers and one index. Vertices and half-edges can therefore be stored in a quadword each, with no or little overhead. The updated algorithm is given in Listing 9.2.

Only pointers pointing towards deleted elements are modified, thus all partly updated triangles will have references to deleted elements (see figure 9.3). Therefore, partly updated triangles will either include a reference to a vertex marked for removal, a half-edge marked as removed, or the three half-edges will not form a loop. If any of these cases are detected, the vertex removal process ignores V, otherwise no conflict is present and the vertex removal continues. Since our cache is not coherent, data may remain in a SPEs cache after its value is updated. Our data structure consistency tests will detect the cases where the data is partly updated and can either ignore the candidate vertex, or dirty the cache and re-read the inconsistent data.

In our test cases, the overall cache-hit ratio is 80–90%. The high cache-hit ratio is due to the fact that geometrically near data elements are likely to be located near each other in memory. In our test cases each triangle is initially defined by three subsequent half-edges. However, for the first vertex and half-edge read for a given candidate vertex V the cache-hit ratio is very low. This is because the candidate vertices are not treated in their natural order, but rather in a sorted order. Thus, the cache-hit ratio is almost unaffected by dirtying the cache at the beginning of each vertex removal operation.

The cost of a cache-miss is reduced if the processor is busy while the memory transfer takes place. This can be achieved by touching the cache before the data is needed. This strategy is only feasible if the program can continue execution before the data is transferred. In our algorithm this is not the case. Bader et al. [5] presented an alternative latency-hiding technique for the Cell BE. They use software-managed threads to let the SPE continue working after a memory request is issued. The SPEs do not have hardware support for quickly switching between threads. Therefore, the program itself is responsible for switching between the software-managed threads. We applied this strategy in our application, manually switching threads after a cache miss. To facilitate this behavior, we implemented two different cache-touching functions

in our software-based cache. One function only starts the required data transfer, while the other function also reports if the data element is already in cache. This allows us to swap threads only when an actual cache-miss is encountered. However, with the high cache-hit ratio in our algorithm this only gave us 2% speedup.

9.3 Results

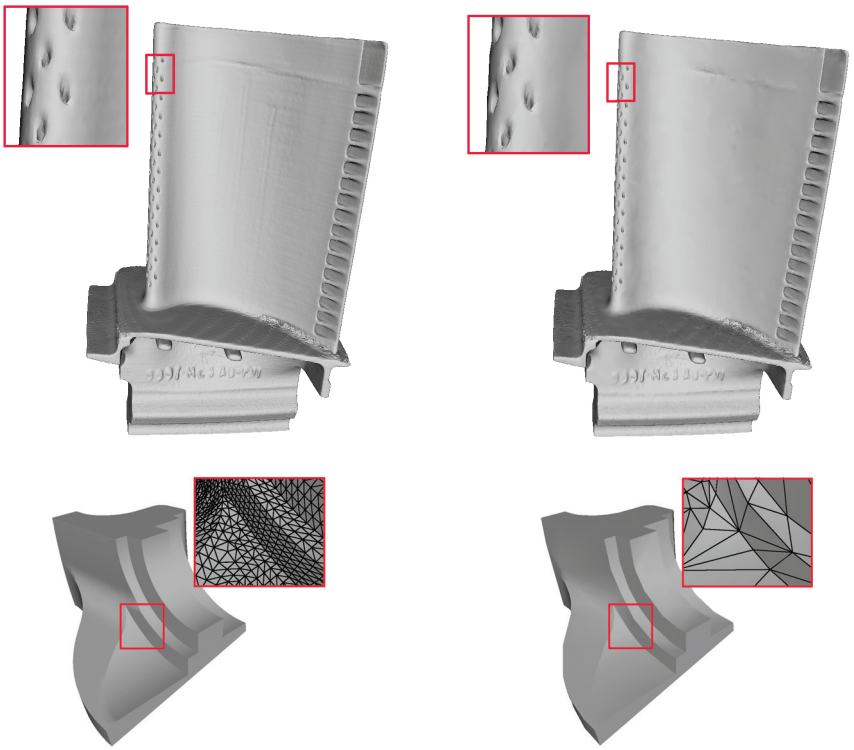


Figure 9.4: Turbine blade (top) and fandisk model (bottom) before and after simplification.

To verify correctness, we implemented a multi-core version, without dynamic load balancing. Since we have not put any effort into optimizing the implementation, we do not include runtimes from the multi-core version. For each parallelizable stage of our algorithm, we uniformly divide the vertices among the threads. The computation time seems to be sufficiently uniform to justify this choice. The same strategy is used for the Cell BE implementation, where the vertices are partitioned, and each SPE is responsible for a subset of the vertices.

Figure 9.4 shows the two test models before and after simplification. The vertex-cost is set to its discrete Gaussian curvature. The 90% vertices with lowest costs are considered as

Table 9.1: Runtimes of simplification, using one SPE on IBM QS21.

model	#vertices	runtime geometric	runtime topology
blade	882954	53.7s	9.0s
fandisk	6475	0.37s	0.05s

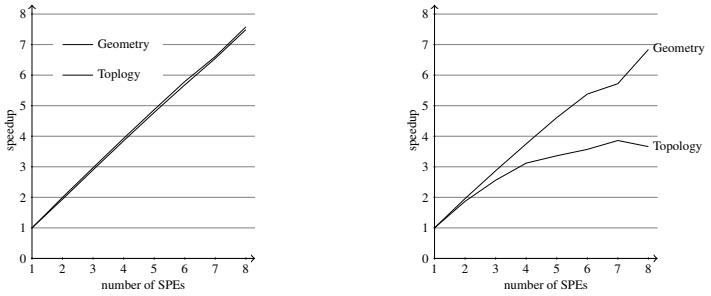


Figure 9.5: Speedup of simplification of the turbine blade model (left) and the fandisk model (right). Benchmark of geometry-based simplification criteria and purely topological ones are presented.

potential for removal. Potential vertices are removed if the maximal angular deviation from the previous model is less than 25 degrees, and all interior angles in the resulting triangles are larger than 10 degrees. This yields good visual quality of the resulting triangulations. However, they do not provide any guaranteed error estimates. These criteria were chosen because of their balance between computations and memory operations. In addition, we include benchmark results where only topology tests are performed. Obviously, the resulting models are not usable, since the geometry is ignored. However, it shows that our algorithm performs well also for less computational intensive decimation criteria.

In the current Cell BE implementation the sorting is performed by the PPE. Parallel sorting algorithms are becoming a standard component for parallel architectures, and is a part of software development kits for GPUs and the standard C++ library distributed with the `gcc` compiler now contains parallel sorting. However, sorting is not yet a part of the API we used for the Cell BE. Implementing parallel sorting algorithms such as AA-sort by Inoue et al. [37] is outside the scope of our work. We therefore performed the benchmarks using a single-threaded sorting algorithm. For the turbine blade model in Figure 9.4, sorting and other sequential parts of the code represents less than one percent of the total runtime when using one SPE. The lack of parallel sorting implementation does not notably affect this benchmark, but must be remedied if the number of cores are to increase greatly.

Figure 9.5 illustrates the speedups achieved using a varying number of SPEs. The corresponding runtimes using one SPE is listed in Table 9.1. The main purpose here, is to highlight how the speedup scales up when the number of SPEs is increasing. For the blade model, the speedup is almost linear for both benchmarks. The much smaller fandisk model does not achieve the same speedup when only topology is tested. This is due to the overhead caused by starting and stopping SPE threads.

9.4 Concluding Remarks

We have developed a flexible framework for shape simplification, suitable for heterogeneous many-core systems. The vertex removal is performed in a thread safe manner, allowing each thread to operate independently and eliminating the need of an independent set of vertices. The flexibility of the framework allows the inclusion of any decimation criterion based on information located inside the one-ring neighborhood. Furthermore, properties associated to the faces can be propagated throughout the simplification process, thus allowing simplification criteria with guaranteed error estimates such as error spheres described in Section 8.3.

Our benchmarks have shown that our framework performs well both for arithmetically intensive and memory intensive decimation criteria. Furthermore, it has sufficiently low overhead to be used in applications treating a large number of smaller models. The use of software-managed threads did not give us the expected performance increase. This may change if the software related to software-managed threads is optimized.

The strategy presented here can be used also for other operations performed on triangulations. Operations such as edge-flip and the Bowyer Watson algorithm for vertex insertion are candidates for this framework.

GPUs supporting the CUDA API perform data transfers of naturally aligned quadwords as atomic operations. Locking mechanisms are now also available on commodity GPUs. This opens up the possibility to develop a GPU-based implementation of our framework. Whether or not such an implementation is feasible is an open question, as the execution units of current GPUs operate in a synchronous manner.

Chapter 10

Conclusions and Perspectives

We have presented and discussed four different algorithms taking advantage of the computational power of heterogeneous computer architectures. The algorithms have targeted different problems, each with different challenges. The main focus has been on topics relevant for mechanical engineering, and our work shows that heterogeneous architectures are well suited for the applications investigated.

During our research period the research field of GPGPU has grown from a research field in its birth into a large research field attracting scientists worldwide. More and more people have opened their eyes both for the research possibilities and the business opportunities emerging from using GPUs for general purpose computations. Being able to take part in this development has been extra interesting and the growth in interest and development of GPUs have been far beyond our expectation. In our opinion, the research in GPGPU and heterogeneous computing has made an important contribution to the commercial interest for using such architectures. The progress in the field has changed the question from "Are GPUs usable for us?" to "How can we use GPUs to improve our applications?". The research in GPGPU has had a large impact on the GPU design today, since it has developed new ways to use the processors, and exposed their strengths and weaknesses.

The CPU and GPU designs are getting more and more similar, as most CPUs are now parallel processors and the GPUs are getting more flexible. In between is the Cell BE processor, including both a traditional CPU core and a number of computational cores intended for data parallelism. We expect more architectures like this, where it is required that software is written to take advantage of different computational resources within the same computer node to obtain high performance. Thus, we expect that there will be an increasing interest for research into developing suitable algorithms, design patterns, and software development tools. It will be very interesting to see how the architectures develop and how they are used.

Bibliography

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS Spring Joint Comput. Conf. 30, Atlantic City*, pages 483–485, 1967.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] E. Audusse, F. Bouchut, M.-O. Bristeau, R. Klein, and B. Perthame. A fast and stable well-balanced scheme with hydrostatic reconstruction for shallow water flows. *SIAM J. Sci. Comput.*, 25(6):2050–2065, 2004.

[4] J. W. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[5] D. A. Bader, V. Agarwal, and K. Madduri. On the design and analysis of irregular algorithms on the Cell processor: A case study of list ranking. In *Proc. of the 21st International Parallel and Distributed Processing Symposium*, pages 1–10. IEEE, 2007.

[6] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

[7] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[8] J. Bolz and P. Schröder. Evaluation of subdivision surfaces on programmable graphics hardware. submitted for publication, 2003.

[9] M. Botsch, D. Bommes, C. Vogel, and L. Kobbelt. GPU-based tolerance volumes for mesh processing. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)*, pages 237–243, Washington, DC, USA, 2004. IEEE Computer Society.

[10] T. Brandvik and G. Pullan. Acceleration of a two-dimensional euler flow solver using commodity graphics hardware. *Proceedings of the I MECH E Part C Journal of Mechanical Engineering Science*, 221:1745–1748, 2007.

[11] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright. Simplification envelopes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 119–128, New York, NY, USA, 1996. ACM.

[12] N. Dadoun and D. G. Kirkpatrick. Parallel algorithms for fractional and maximal independent sets in planar graphs. *Discrete Appl. Math.*, 27(1-2):69–83, 1990.

[13] C. DeCoro and N. Tatarchuk. Real-time mesh simplification using the GPU. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 161–166, New York, NY, USA, 2007. ACM.

[14] T. Dokken, T. Hagen, and J. Hjelmervik. An introduction to general-purpose computing on programmable graphics hardware. In *Geometric Modelling, Numerical Simulation, and Optimization: Applied Mathematics at SINTEF*, pages 123–161. Springer Verlag, 2007.

[15] C. Dyken, M. Reimers, and J. Seland. Real-time GPU silhouette refinement using adaptively blended bézier patches. *Computer Graphics Forum*, 27(1):1–12, Mar. 2008.

[16] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[17] N. C. for Atmospheric Research (NCAR), NCEP, FSL, AFWA, and FAA. The Weather Research & Forecasting Model. http://www.wrf-model.org.

[18] G. Foucault, J.-C. Cuillière, V. François, J.-C. Léon, and R. Maranzana. Adaptation of cad model topology for finite element analysis. *Comput. Aided Des.*, 40(2):176–196, 2008.

[19] G. Foucault, P. Marin, and J.-C. Léon. Mechanical criteria for the preparation of finite element models. In *Int. Meshing Roundtable, Williamsburg (USA), 20-22 September*, pages 413–426, 2004.

[20] M. Franc and V. Skala. Parallel triangular mesh decimation without sorting. In *SCCG '01: Proceedings of the 17th Spring conference on Computer graphics*, page 22, Washington, DC, USA, 2001. IEEE Computer Society.

[21] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH*, pages 264–270, 1997.

[22] M. Geimer and O. Abert. Interactive ray tracing of trimmed bicubic bézier surfaces without triangulation. In *WSCG (Full Papers)*, pages 71–78, 2005.

[23] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 102–111, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[24] C. Gotsman, S. Gumhold, and L. Kobbelt. Simplification and compression of 3d meshes. In *Tutorials on Multiresolution in Geometric Modelling*, pages 319–361. Springer, 2002.

[25] M. Guthe, Ákos Balázs, and R. Klein. GPU-based trimming and tessellation of NURBS and T-Spline surfaces. *ACM Trans. Graph.*, 24(3):1016–1023, 2005.

[26] T. R. Hagen, M. O. Henriksen, J. M. Hjelmervik, and K. A. Lie. Using the graphics processor as a high-performance computational engine for solving systems of hyperbolic conservation laws. In *Geometric Modelling, Numerical Simulation, and Optimization: Applied Mathematics at SINTEF*, pages 211–264. Springer Verlag, 2007.

[27] T. R. Hagen, J. M. Hjelmervik, K.-A. Lie, J. R. Natvig, and M. O. Henriksen. Visual simulation of shallow-water waves. *Simulation Practice and Theory. Special Issue on Programmable Graphics Hardware*, 13(9):716–726, 2005.

[28] T. R. Hagen, K.-A. Lie, and J. R. Natvig. Solving the euler equations on graphics processing units. In V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 220–227. Springer, 2006.

[29] O. Hamri. *Method, models and tools for Finite Element model preparation integrated into a product development process*. PhD thesis, co-tutelle between Institut National Polytechnique de Grenoble and University of Genova, 2006.

[30] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 109–118, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[31] P. S. Heckbert and M. Garland. Multiresolution surface modeling course siggraph 97, 1997.

[32] J. Hjelmervik and J.-C. Léon. GPU-accelerated shape simplification for mechanical-based applications. In *Shape Modeling International*, pages 91–102. IEEE Computer Society, 2007.

[33] J. M. Hjelmervik and T. R. Hagen. GPU-based screen space tessellation. In M. Dæhlen, K. Mørken, and L. L.Schumaker, editors, *Mathematical Methods for Curves and Surfaces: Tromsø 2004*, pages 213–221. Nashboro Press, 2005.

[34] J. M. Hjelmervik, T. R. Hagen, and J. Seland. shallows, 2005. http://shallows.sourceforge.net.

[35] H. Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 189–198, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[36] T. J. Hughes, J. Cottrell, and Y. Bazilevs. Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement. *Computer Methods in Applied Mechanics and Engineering*, 194:4135–4195, 2005.

[37] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society.

[38] T. Kanai and Y. Yasui. Surface quality assessment of subdivision surfaces on programmable graphics hardware. In *Proc. International Conference on Shape Modeling and Applications 2004*, pages 129–136, Los Alamitos, CA, 2004. IEEE CS Press.

[39] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 266–272, New York, NY, USA, 1984. ACM.

[40] L. Kobbelt, S. Campagna, and H. peter Seidel. A general framework for mesh decimation. In *in Proceedings of Graphics Interface*, pages 43–50, 1998.

[41] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[42] A. Kurganov and D. Levy. Central-upwind schemes for the Saint-Venant system. *M2AN Math. Model. Numer. Anal.*, 36(3):397–425, 2002.

[43] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55, New York, NY, USA, 2001. ACM.

[44] P. Lax and B. Wendroff. Systems of conservation laws. *Comm. Pure Appl. Math.*, 13:217–237, 1960.

[45] A. W. F. Lee, W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin. MAPS: multiresolution adaptive parameterization of surfaces. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 95–104, New York, NY, USA, 1998. ACM.

[46] R. LeVeque. *Finite volume methods for hyperbolic problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, 2002.

[47] P. Lindstrom. Out-of-core simplification of large polygonal models. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 259–262, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[48] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118, New York, NY, USA, 1996. ACM.

[49] F. Losasso, H. Hoppe, S. Schaefer, and J. D. Warren. Smooth geometry images. In L. Kobbelt, P. Schröder, and H. Hoppe, editors, *Symposium on Geometry Processing*, volume 43 of *ACM International Conference Proceeding Series*, pages 138–145. Eurographics Association, 2003.

[50] J. Michalakes and M. Vachharajani. GPU acceleration of numerical weather prediction. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–7, 2008.

[51] S. J. Owen, D. R. White, and T. J. Tautges. Facetbased surfaces for 3-d mesh generation. In *Proc. 11 th Int. Meshing Roundtable*, pages 297–311, 2002.

[52] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[53] H.-F. Pabst, J. Springer, A. Schollmeyer, R. Lenhardt, C. Lessig, and B. Froehlich. Ray casting of trimmed nurbs surfaces on the GPU. *rt*, 0:151–160, 2006.

[54] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[55] M. Reimers and J. Seland. Ray casting algebraic surfaces using the frustum form. *Computer Graphics Forum*, 27:361–370(10), April 2008.

[56] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. In B. Falcidieno and T. L. Kunii, editors, *Geometric Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, 1993.

[57] M. Rumpf and R. Strzodka. Using graphics cards for quantized FEM computations, 2001.

[58] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 146–155, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[59] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 65–70, New York, NY, USA, 1992. ACM.

[60] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[61] R. Strzodka and D. Göddeke. Mixed precision methods for convergent iterative schemes. In *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*, pages D–59–60. UNC Chapel Hill, May 2006.

[62] J. Tessendorf. Simulating ocean water. SIGGRAPH 2004 Course Notes.

[63] Top 500 supercomputer sites. http://www.top500.org/lists/2008/06, June 2008.

[64] S. Venkataraman, M. Sohoni, and G. Elber. Blend recognition algorithm and applications. In *SMA '01: Proceedings of the sixth ACM symposium on Solid modeling and applications*, pages 99–108, New York, NY, USA, 2001. ACM.

[65] P. Véron and J.-C. Léon. Shape preserving polyhedral simplification with bounded error. *Computers & Graphics*, 22(5):565–585, 1998.

[66] Y. Zhao, F. Qiu, Z. Fan, and A. Kaufman. Flow simulation with locally-refined lbm. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 181–188, New York, NY, USA, 2007. ACM.

Appendix A

Shallows

`Shallows` is an open source programming library distributed under GPL. It is developed to make GPGPU programming easier and safer. The main objective was to lift the burden of dealing with an API designed for rendering when developing GPGPU applications. Since we were developing advanced rendering techniques as well as GPGPU algorithms, it was important that `Shallows` could be used both for visualization and computation. It can easily be integrated with existing OpenGL applications. Performance is the main objective for using the GPU for computations, therefore `Shallows` does not generate shaders, but is rather an abstraction layer over a OpenGL. `Shallows` consists of intuitive C++ classes, that encapsulate the functions most commonly used in GPGPU applications. Therefore, the use of this library does not impose any performance penalty for the rendering.

A.1 Overview

`Shallows` is a C++ abstraction to the programmable parts of the graphical pipeline. We found it most intuitive to center the library around a `Program` class, containing enough data members and functionality to run a GPU program. A `Program` object can contain a vertex shader and/or a fragment shader. This corresponds to the definition of program and shader by the OpenGL specification [60]. In addition a `Program` object contains references to input textures and output render targets. Other parameters to the shaders can be set using member functions defined in the `Program` class, but are maintained directly by OpenGL. A feature of `Shallows` is to extend the error checking by testing it the parameter types match the ones defined in the shader.

Calling the `run` function on a `Program` object initiates the rendering needed to perform the computations. Before the actual rendering takes place, `Shallows` ensures that the associated textures and render targets are used. To facilitate this, textures, render targets, and frame buffers are encapsulated in C++ classes. This also enables `Shallows` to take responsibility for creating and freeing these object types. Shared pointers are used to determine when the objects should be freed.

A.2 Usage Example

Many algorithms are stencil updates that consist of a discrete set of cells and a computational kernel which is to be invoked for each cell to compute the new cell value, for example applications in physical simulation and image processing. A typical CPU implementation would consist of a construction that explicitly loops over all cells and calls the kernel for each of them. Since the programmable fragment and vertex processors have replaced their fixed-function counterparts, the shaders cannot be called explicitly, but are instead invoked by the graphics pipeline.

Here, we will present an implementation of a simple, yet illustrative example of general-purpose computing on a GPU, namely the simulation of heat conduction using the linear heat equation. Visually, this corresponds to repeatedly applying a Gaussian blur to an image. For this example we use `Shallows` together with OpenGL Shading Language.

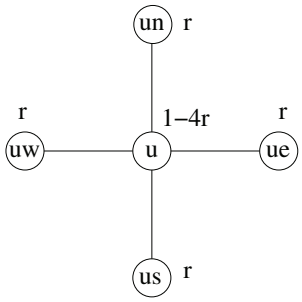


Figure A.1: The stencil used for simulating heat conduction.

The 2D heat equation describes the transport of heat in a body

$$u_t = u_{xx} + u_{yy}.$$

Using standard finite differences, this equation is easily discretized as

$$U_{i,j}^{n+1} = (1 - 4r)U_{i,j}^{n} + r\left(U_{i-1,j}^{n} + U_{i,j-1}^{n} + U_{i+1,j}^{n} + U_{i,j+1}^{n}\right), \tag{A.1}$$

where h is the grid size, k is the length of the time step, and $r = k/h^2$. This discretization requires that $r =\leq 1/4$ to guarantee stability, thus $k = h^2/4$ is the largest time step that provides a correct solutions. From (A.1) we see that the value of a grid cell can be computed by combining the values of the neighboring cells and the value of the cell itself at the previous time step, yielding the stencil illustrated in Figure A.1.

Listing A.1 contains the C++ source code used in the simulation of heat conduction. First, we allocate the render targets we use as computational domain. We require one off-screen frame buffer containing two render targets, one for holding the state at the current time step U^{n+1} and one for the previous time step U^n.

```
fb.reset(new OffScreenBuffer(BUFF_DIM, BUFF_DIM));
rt[0]=fb->createRenderTexture2D();
rt[1]=fb->createRenderTexture2D();
```

(a) Original image. (b) Image after 100 iterations. (c) Image after 500 iterations.

Figure A.2: Repeated Gaussian blur applied to a 512x512 image.

Throughout the simulation we "ping-pong" between these two buffers, so that we always read from the most recently updated render target. After allocating these render targets, we load the fragment shader given in Listing A.2. This will make `Shallows` compile and link the source code, viz:

```
heatProg.readFile("heat.shader");
```

Then, we set the output frame buffer:

```
heatProg.setFrameBuffer(fb);
```

The fragment shader takes the distance between grid cells as a uniform parameter, which we set with:

```
heatProg.setParam1f("h", 1.0/BUFF_DIM);
```

`Shallows` can set the texture coordinates to be in any range. The range is set to $[0, 1]$ by calling the function `useNormalizedTexCoords`. The rasterization will interpolate the texture coordinate, so that the texture coordinate sent to the fragment shader is $((i+0.5)*h, (j+0.5)*h)$ for cell u_{ij}. The final step before starting the simulation is to set up the initial state. This is performed by filling a floating-point array with the initial state, and copying it to a texture, by the following function call:

```
fillRGBATexture(rt[1]->getTexture(), p);
```

At the beginning of each iteration of the simulation we set the most recently updated render target to be used as texture. The output target of the program is then set to be the other render target.

```
heatProg.setInputTexture("texture",
                         rt[(i+1)%2]->getTexture());
heatProg.setOutputTarget(0, rt[i%2]);
```

Listing A.1: Finite-difference solver for the heat equation using `Shallows`.

```
shared_ptr<OffScreenBuffer> fb;
fb.reset(new OffScreenBuffer(BUFF_DIM, BUFF_DIM));

shared_ptr<RenderTexture2D> rt[2];
rt[0]=fb->createRenderTexture2D();
rt[1]=fb->createRenderTexture2D();

GLProgram heatProg;
heatProg.readFile("heat.shader");
heatProg.setFrameBuffer(fb);
heatProg.setParam1f("h", 1.0/BUFF_DIM);
heatProg.useNormalizedTexCoords();

float *p;
  /*
   * fill p with initial state
   */
fillRGBATexture(rt[1]->getTexture(), p);

/* Start the simulation */
for (int i=0; i<MAX_ITERATIONS; i++) {
  heatProg.setInputTexture("texture", rt[(i+1)%2]->getTexture());
  heatProg.setOutputTarget(0, rt[i%2]);
  heatProg.run();
 }
```

Listing A.2: Fragment and vertex shader for the heat equation written in GLSL.

```
[Vertex shader]

void main ()
{
  // Pass vertex position to fragment shader
  gl_Position = gl_Vertex;
  // Pass texture coordinate to fragment shader
  gl_TexCoord[0]=gl_MultiTexCoord0;
}

[Fragment shader]

uniform sampler2D texture; // Texture containing previous timestep
uniform float h; // Distance between grid cells (in the texture)

void main ()
{
  const float r = 0.25;

  vec4 u = texture2D(texture, gl_TexCoord[0].xy);
  vec4 ue = texture2D(texture, gl_TexCoord[0].xy + vec2(h, 0.0));
  vec4 uw = texture2D(texture, gl_TexCoord[0].xy - vec2(h, 0.0));
  vec4 un = texture2D(texture, gl_TexCoord[0].xy + vec2(0.0, h));
  vec4 us = texture2D(texture, gl_TexCoord[0].xy - vec2(0.0, h));

  vec4 result = (1.0-4.0*r)*u + r*(ue + uw + un + us);
  gl_FragColor = result;
}
```

To execute the fragment shader, we call the `run` function in the `GLProgram` object, which renders a geometric primitive covering the viewport. This function assumes that the transformation matrix is an identity matrix. The vertex shader in Listing A.2 therefore does not transform the vertex position, only passes it through the rasterization (for interpolation) into the fragment shader.

From (A.1) we get the stencil illustrated in Figure A.1, and a fragment shader that implements this scheme is given in Listing A.2. The shader first reads the cell value from the texture containing the previous time step, followed by reading the neighboring cell values. Note that the texture coordinates are floating-point numbers in the range $[0, 1]$, and that the distance between two data cells therefore is h. The values read are combined according to (A.1), the result is returned as the new color of the fragment, by writing it to the variable `gl_FragColor`.